Systèmes d'exploitation

ETE8-IFON1

Philippe SWARTVAGHER
ph-sw.fr

CC BY NC ND

BORDEAUX INP Enseirb-Matmeca

À propos de ce cours

Organisation

- 8 séances de CM ($8 \times 1h20$)
- 2 séances de TP ($2 \times 2 \times 1h20$)

Objectifs

- Connaître les concepts de base du fonctionnement d'un système d'exploitation
- Mieux comprendre le fonctionnement d'un système Linux
- Acquérir un peu de culture autour des OS

Évaluation

- Partiel

Pré-requis

- Programmation C (S5)
- Systèmes d'exploitation (S5)
- Programmation système (S7)

Postionnement de ce cours

- En-dessous de la programmation système
- Au-dessus de l'architecture du processeur

- Orienté Linux
- Orienté serveur
- Orienté administration système

- Identifier les problèmes auxquels un système d'exploitation est confronté
- Réfléchir aux solutions possibles
- Cours d'OS au S5 : « qu'est-ce qu'il est possible de faire avec un OS » ; ce cours au S7 : « comment les fonctionnalités peuvent être implémentées »

Pourquoi étudier les systèmes d'exploitation ?

- Logiciels parmi les plus complexes
  - Très technique (le plus proche possible du matériel !)
  - Génie logiciel (beaucoup de code, de composants, d'interfaces, ...)
  - Optimisations (performances cruciales)
  - Des compromis
  - Algorithmique, heuristiques, ...
- Permet de mieux comprendre le fonctionnement d'un ordinateur et les problèmes qui peuvent arriver
- On retrouve des questions similaires ailleurs (navigateurs web, machine virtuelle Java, ...)

- Très intéressant !

Ressources et crédits

Ce cours est basé, entre autres, sur :

- Le cours de Brice Goglin : https://people.bordeaux.inria.fr/goglin/teaching/Systeme.html
- Le cours de Raymond Namyst : https://gforgeron.gitlab.io/se/cours/
- Le cours de Ludovic Courtès : https://people.bordeaux.inria.fr/lcourtes/
- Le cours de Martin Quison : https://mquinson.frama.io/ensr-progsys/aGr3g/

Ouvrages de référence :

- *Systèmes d'exploitation*, Andrew S. Tanenbaum
- *Understanding the Linux Kernel*, Daniel P. Bovet & Marco Cesati

- Construisez votre propre distribution Linux : https://linuxfromscratch.org
- Tutoriels en ligne pour créer son propre noyau

À la fin du cours, vous ne devriez pas être perdus en lisant :

- https://peteris.rocks/blog/htop/
- https://www.0xkato.xyz/linux-process-memory/

Sommaire

Avant de commencer...

Concepts généraux

Que se passe-t-il au démarrage ?

Exécution par le processeur

Processus et threads

Ordonnancement

Mémoire

Stockage

Virtualisation

Ce qui ne rentrait pas dans les autres sections

Avant de commencer...

Quelques rappels sur l'architecture d'un ordinateur

- Un ou plusieurs processeurs (avec une architecture spécifique) ...
- ... découpés en cœurs (physiques) ...
  - Chaque cœur exécute indépendamment son flot d'instructions
  - Parfois découpés en cœurs logiques (Hyper-threads, SMT [1])
- ... qui accèdent à de la mémoire RAM [2] ...
  - Mais avec différents niveaux de cache avant
- ... auxquels sont reliés des périphériques (I/O [3]) via un bus PCI [4] :
  - Disque dur mécanique (HDD [5]), SSD [6], NVMe [7], ...
  - Carte réseau (NIC [8])
  - Carte graphique (GPU [9])
  - Contrôleur USB [10]
  - ...

1. Simultaneous Multithreading
2. Random Access Memory
3. Input/Output
4. Peripheral Component Interconnect
5. Hard Disk Drive
6. Solid-State Drive
7. Non-Volatile Memory Express
8. Network Interface Card
9. Graphics Processing Unit
10. Universal Serial Bus

Architectures processeur

- « *Comment parler au processeur ?* »
- On parle aussi de *jeu d'instructions* ou ISA [11]
- Chaque architecture a son language assembleur
  - Plus ou moins de fonctionnalités et d'expressivité (RISC [12] *vs* CISC [13])
  - Réflète aussi le fonctionnement matériel du processeur

⇒ Il faut compiler les programmes pour chaque architecture !

  - Compiler pour une autre architecture : *cross*-compiler

- Exemples :
  - x86_64 : la majorité des serveurs, ordinateurs, ...
  - ARM : téléphones, tablettes, ordinateurs Apple, certains ordinateurs, ...
  - PowerPC
  - RISC-V
  - ...

11. Instruction Set Architecture
12. Reduced Instruction Set Computer
13. Complex Instruction Set Computer

Quelques ordres de grandeurs - Exécution

Processeur

- Fréquence : $\sim$ 3 GHz
  - = Nombre de cycles par seconde (à 3 GHz, 1 cycle = 0,3 ns)
  - Limitée par la finesse de gravure des transistors, la consommation électrique et la chaleur dégagée
  - Les durées suivantes en découlent
- Dizaine(s) de cœurs
- Instruction : 1 ns
  - Instruction assembleur
  - Une instruction peut nécessiter plusieurs cycles
- Instruction atomique : 100 ns
- Appel système : 50-1000 ns

Quelques ordres de grandeurs - Mémoire et I/O

	Composant	Capacité	Temps d'accès	Débit	Comparaison géographique du temps d'accès
	Registre	$\mathcal{O}(1)$ octet	1 cycle		Bordeaux
	Caches	10 Ko - 100 Mo	1-10 ns		Talence
	RAM	$\mathcal{O}(10)$ Go	50-100 ns	50 Go/s	Toulouse
	Bus PCI		$>$ 100 ns	Go/s	
Disque	SSD/NVMe	0,1 - 10 To	1-100 µs	0,1-10 Go/s	Lune
Disque	Mécanique	10 To	10 ms	10-100 Mo/s	Mars
Réseau	Grand public		10-1000 ms	100 Mo/s	
Réseau	HPC [14]		1 µs	10-50 Go/s	

14. *High Performance Computing*

Concepts généraux

Qu'est-ce qu'un système d'exploitation ?

- SE : *Système d'Exploitation*
- OS : *Operating System*

- Entre le matériel et les applications utilisateur
- Fournit une abstraction du matériel aux applications
  - Cache les détails du matériel
- Gère de multiples tâches et de multiples utilisateurs

- Parfois on n'en utilise pas (systèmes embarqués...)

Ceci n'est **pas** le système d'exploitation

- Compilateurs
- Systèmes de fenêtrage (X, Wayland)
- Environnements de bureau (GNOME, Xfce, KDE Plasma, Cinnamon, ...)
- Interpréteurs de commandes (Bash, Zsh, Fish, ...)
- Outils en ligne de commande (`ls`, `cp`, ...)
- Un super-utilisateur (`root`)
- Un processus (au sens système) particulier

Ceci **peut** être le système d'exploitation

- Le noyau (*kernel*)
  - S'exécute en mode privilégié par le processeur
- Quelques bibliothèques autour, comme les bibliothèques standards
  - libc sous Linux
  - Fournit notamment les appels systèmes
- Éventuellement des outils, services, démons, ...
  - Exemple : un client DNS

- Finalement, les limites d'un système d'exploitation sont un peu floues...

Root *vs* noyau

Root

- Super-**utilisateur**
- Utilisateur qui a un accès privilégié au noyau, avec plus de droits
- Privilégié **logiciellement** par le **noyau**

Noyau

- Contrôle tout, peut accèder à tout
- A un accès privilégié à tout le matériel
- Privilégié **matériellement** par le **processeur**

Que serait un système d'exploitation primitif ?

Il nécessiterait les hypothèses suivantes :

- Seulement l'abstraction du matériel, pas de protection
- Un unique utilisateur
- Un unique programme s'exécute à chaque instant
- Les programmes n'ont pas de bugs, ne sont pas malintentionnés

$\Rightarrow$ MS-DOS (1981-2001)

Problèmes traités par un système d'exploitation

Multi-tâches

- Plusieurs programmes s'exécutent en même temps
  - Avec chacun sa propre (zone) mémoire !
  - On discutera de ce qui signifie vraiment « *en même temps* »
- Si un programme doit attendre, un autre s'exécute

Protection et isolation

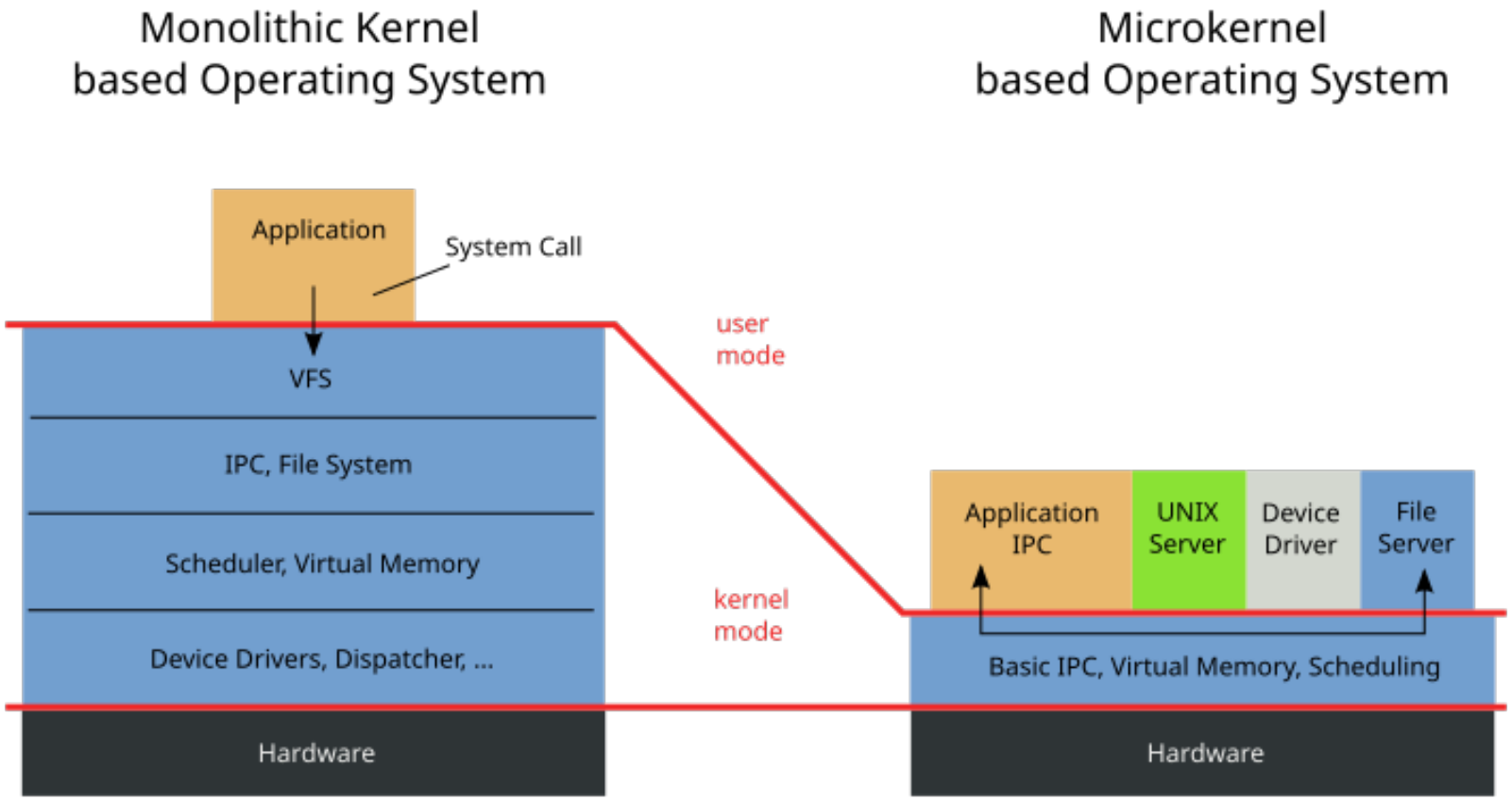
- Traiter le cas d'un programme qui se comporte mal :
  - Boucle infinie
  - Accède à de la mémoire qui n'est pas destinée à ce programme

Multi-utilisateurs

- Plusieurs utilisateurs simultanés possibles, exécutant des programmes différents
- Besoin d'une répartition équitable des ressources entre utilisateurs

Structures des systèmes d'exploitation

- Comment organiser le code d'un système d'exploitation ?
- Deux grandes familles :
  - Noyau monolithique
  - Micro-noyau



Noyau monolithique *vs* micro-noyau

Noyau monolithique

- Tout le code noyau est exécuté en mode privilégié
- Toutes les fonctionnalités sont très liées
- Vue globale de toutes les ressources : ordonnancement facilité
- Difficulté d'étendre les fonctionnalités
- Modification du noyau : redémarrage nécessaire
- Plus grande surface d'attaque
- Un bug dans un composant peut affecter tout le noyau

Micro-noyau (+ *serveurs*)

- Le code du noyau est réduit au minimum
- Fonctionnalités séparées dans des processus nommés *serveurs*, exécutés en mode utilisateur
- Conséquences d'un bug limitées
- Extensible plus facilement
- Problème de performances ?

- $\Rightarrow$ Pour un noyau monolithique, l'étude des systèmes d'exploitation revient surtout à l'étude des noyaux

Quelques systèmes d'exploitation

Les plus courants / connus / utilisés

- Windows
- Linux
- MacOS
- Android / iOS
- FreeBSD, NetBSD, OpenBSD

Un peu plus marginaux ou anciens

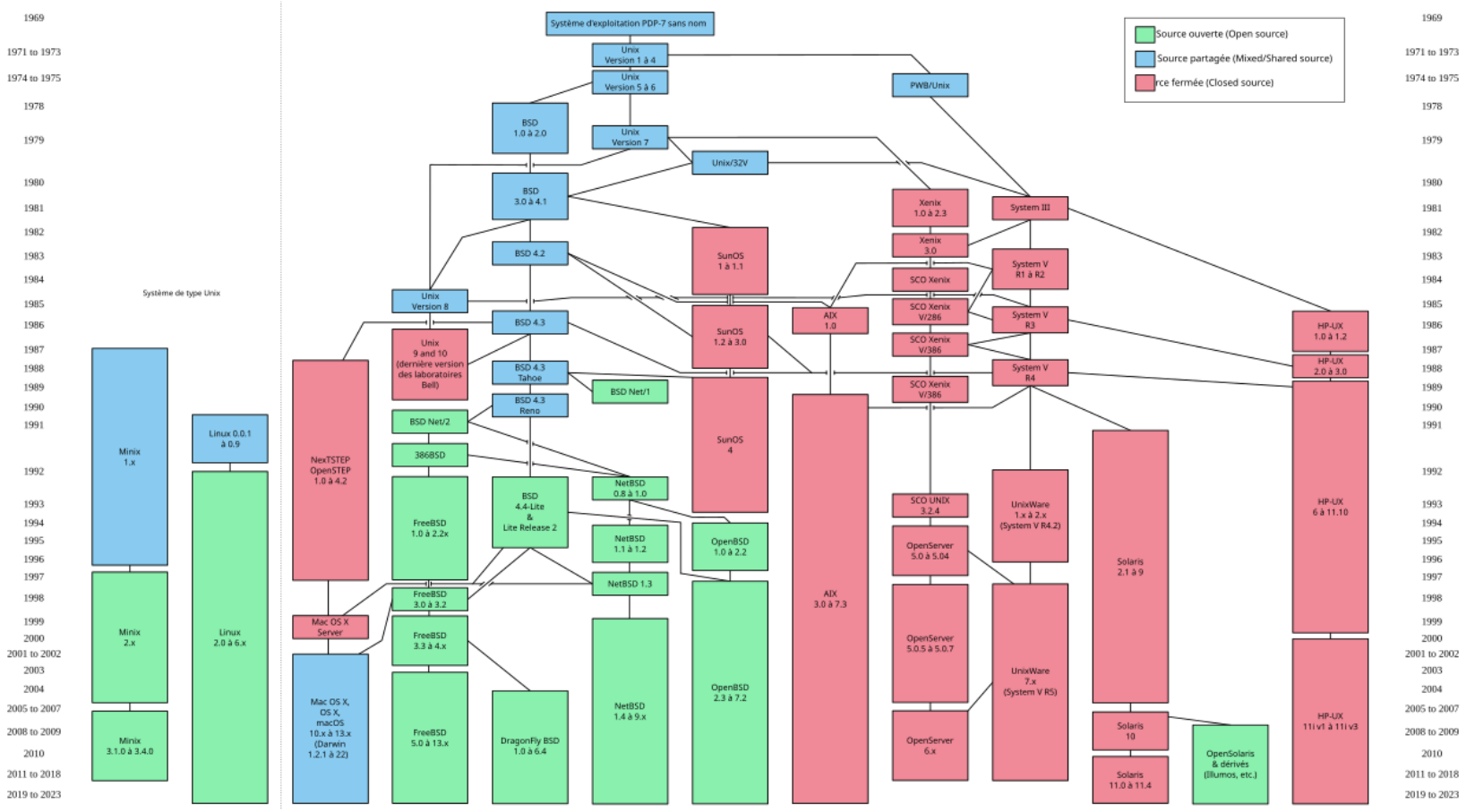
- Solaris
- GNU/Hurd : le seul vrai OS avec micro-noyau ?
- Plan9

Linux *vs* GNU/Linux *vs* *BSD

- Linux
  - Noyau créé par Linus Torvalds en 1991
- GNU
  - Projet de système d'exploitation créé par Richard Stallman en 1983
  - Noyau Hurd et tout un ensemble de logiciels [15]
  - Problème : le noyau Hurd n'était pas prêt
  - ⇒ Utilisation de Linux comme noyau
- GNU/Linux
  - Ensemble formé des outils GNU (Glibc, coreutils, ...) et du noyau Linux
- *BSD
  - *Berkley Software Distribution*
  - Dérivés de Unix
  - Ensembles noyau et logiciels pour manipuler le système
  - Ressemble beaucoup à (GNU/)Linux
  - Beaucoup utilisé pour les serveurs et les équipements réseau
  - FreeBSD : généraliste
  - NetBSD : compatible avec beaaaauuuucooouuup d'architectures
  - OpenBSD : accent sur la sécurité (développe également OpenSSH, OpenBGPD, OpenNTPD, LibreSSL, ...)

15. https://www.gnu.org/manual/blurbs.html

L'arbre généalogique



https://fr.wikipedia.org/wiki/Unix

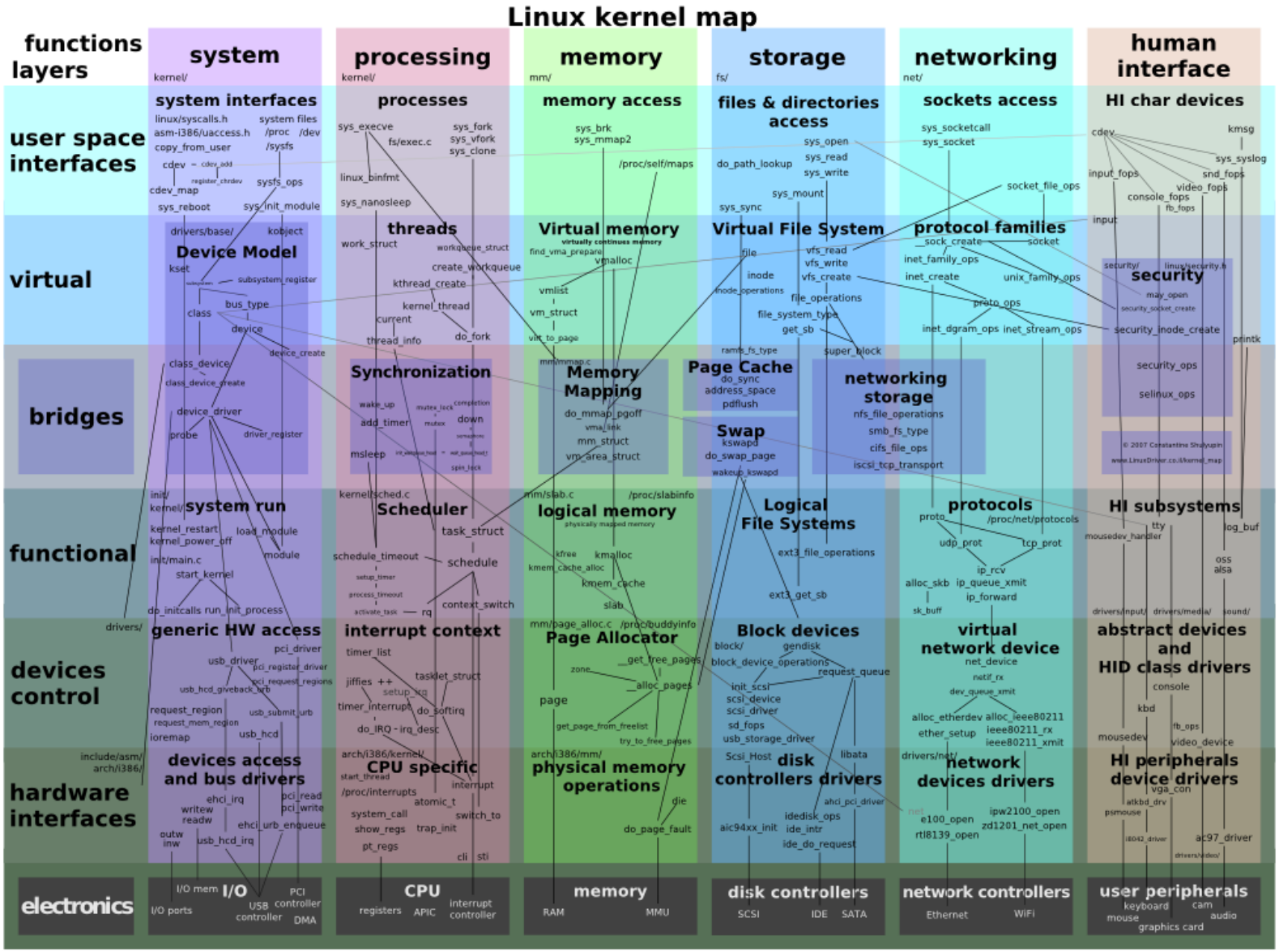
Linux : un noyau monolithique modulaire

Modulaire :

- Chargement / déchargement dynamique de code optionnel
  - Pilotes de périphériques
  - Fonctionnalités spécifiques
- Évite d'augmenter encore la taille du noyau
- Pas besoin de redémarrage pour faire évoluer certaines fonctionnalités du noyau
- Diffusion des modules noyau éventuellement de façon indépendante du noyau

- Exemple jouet classique : créer un module qui crée un type de périphérique qui écrit toujours la même chose

Composants du noyau Linux



https://fr.wikipedia.org/wiki/Noyau_Linux

Et les distributions ?

- Debian, Ubuntu, ArchLinux, Guix, Gentoo, Fedora, CentOS, ... [16]
- A du sens surtout dans le monde Linux
- Fournit un ensemble cohérent de logiciels, configurés pour fonctionner ensemble
  - Un noyau
  - Une bibliothèque C
  - Un gestionnaire de services
  - Des environnements de bureau
  - Une collection de logiciels utilisateurs (Firefox, LibreOffice, VLC, ...)
- Un outil pour la gestion des logiciels (installation, mise à jour)
- Différences surtout dans la gestion des paquets
- Matériel, usage ciblés, philosophie, ... parfois spécifiques
- Politique de mise à jour (versions, *rolling release* ?)
- Fait des choix techniques
  - Quelles architectures supporter ?
  - Quels logiciels par défaut ?

16. https://fr.wikipedia.org/wiki/Liste_des_distributions_GNU/Linux

Que se passe-t-il au démarrage ?

Que se passe-t-il lors du démarrage ?

Mise sous tension

0. Les cœurs du processeur ne savent pas quoi exécuter, la mémoire est vide
1. Le premier cœur (*Bootstrap Processor*) sait à quelle adresse lire le code à exécuter
   - Adresse vers une mémoire spéciale (ROM, non volatile)

Que se passe-t-il lors du démarrage ?

BIOS

2. Le processeur exécute le BIOS [17]
   - Ou plus récemment UEFI [18]
   - Interface bas-niveau avec le matériel
   - Détecte les composants
   - Sait faire plus ou moins de choses
     - Configuration du matériel
     - Exécution de code depuis différents endroits (disque dur, USB, réseau, ...)
3. Charge le code de l'OS
   - Depuis le réseau
   - Depuis un disque, en regardant le MBR [19] ou plus récemment la table des partitions GPT [20]
4. Exécution du code de l'OS

17. Basic Input Output System
18. Unified Extensible Platform Interface
19. Master Boot Record
20. GUID Partition Table

Que se passe-t-il lors du démarrage ?

Boot Loader

3. Le BIOS peut charger un Boot Loader au lieu d'un OS
   - GRUB [21]
   - Mini-OS pour choisir quel OS lancer et avec quels paramètres
   - Sur quel disque se situe l'OS...
4. Chargement et exécution du code de l'OS

21. GRand Unified Bootloader

Que se passe-t-il lors du démarrage ?

Exécution de l'OS

5. Démarrage du noyau
   - Détection des ressources matériels par le BIOS
   - Création des structures de données internes au noyau
6. Démarrage des autres cœurs
7. Lancement de quelques démons noyau
8. Lancement du premier processus utilisateur : l'*init*
   - Aujourd'hui : systemd majoritaire sous Linux
   - Mais aussi : OpenRC, runit, launchd pour MacOS, GNU Shepherd
   - Gére tous les processus utilisateurs en arrière-plan, lance le gestionnaire de session, ...
   - PID 1

Exécution par le processeur

Exécution dans les processeurs

Exécution des instructions

- Une instruction par cycle par pipeline
- Plusieurs pipelines dans chaque cœur
- Durée du cycle reliée à la fréquence du processeur
  - La fréquence varie selon la charge, la température, l'utilisation de certaines instructions (vectorielles)

Contexte d'exécution

- Registres utilisateurs :
  - Données
  - Adresses : pile, tas, données, ...
- Registres de statut et de contrôle :
  - *Program Counter* (PC)
  - Statut de l'unité arithmétique flottante

Modes d'exécution

- Plusieurs niveaux de privilèges
- Mis en place matériellement par le processeur
- Exemples d'opérations restreintes :
  - Accès à certains registres de contrôles
  - Instructions d'entrée/sortie bas-niveau
  - Gestion de la mémoire (changer la table des pages)
  - Accès à certaines zones mémoires
- Le noyau peut tout faire : exécuté en *Ring 0* sur x86
- L'utilisateur est limité : exécuté en *Ring 3* sur x86
  - Pas d'accès à toute la mémoire, encore moins la mémoire du noyau
  - Pas d'accès bas-niveau au matériel

Exceptions et interruptions

- Evénements inattendus qui interrompent temporairement l'exécution en cours
- Le cœur saute immédiatement à un traitant, en mode privilégié
  - Fonction définie par le noyau au démarrage
  - Dépend du type d'exception / interruption
- Retour ensuite à l'exécution initiale

Exception

- Interruption par le processeur lui-même
- Uniquement quand le processeur travaille : nécessite un contexte d'exécution
- En cas d'erreur : division par zéro, mauvais accès mémoire, ...
- Comment continuer ? Un traitant va (essayer de) corriger l'erreur
  - Succès : reprise de l'exécution
  - Echéc : le processus est tué

Mécanisme utilisé pour le projet de DSM :
l'accès à une page qui ne nous appartient provoque une erreur de segmentation, on corrigeait le problème dans le traitant de l'exception.

Interruptions

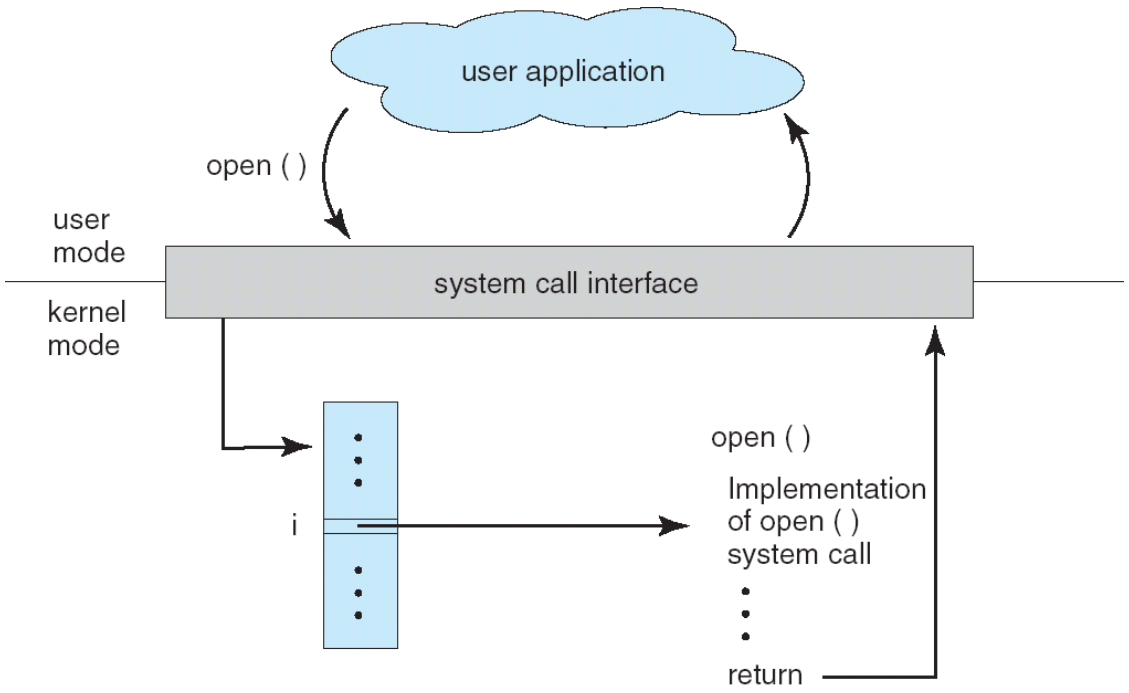
- IRQ : *Interrupt Request*
- Message d'un périphérique
  - Signal éléctrique sur une broche du processeur
  - Écriture à une adresse précise
- Possible n'importe quand : dépend de l'activité des périphériques
  - Déplacement de la souris
  - Un paquet vient d'arriver sur la carte réseau
  - Le disque vient de finir une lecture ou écriture
- Traitant dans le pilote du périphérique
- Les applications ne se rendent compte de rien

- Interruption particulière : interruption régulière de l'horloge
  - Permet de passer la main régulièrement au noyau
  - Environ 1 fois par milliseconde sous Linux

- Regardez le contenu de `/proc/interrupts`

Une exception particulière : l'appel système

- Exception forcée par l'application
- Pour effectuer une opération privilégiée
- Passage en mode noyau
- Se produit uniquement depuis l'espace utilisateur
- Traité dans le contexte d'un processus, mais en mode privilégié



Fonctionnement d'un appel système

Architecture x86 en 32 bits :

1. Numéro de l'appel système dans le registre `%eax`
2. Paramètres dans les registres `%ebx`, `%ecx`, `%edx`, `%esi` et `%edi`
3. Instruction assembleur dédiée (`syscall`)
4. Sauvegarde des paramètres et du contexte utilisateur
5. Passage en mode privilégié dans un traitant dans le noyau
6. Saut à `syscall_table[numéro]`
7. Vérification des paramètres, des droits, ...
8. Exécution du code de l'appel système
9. Valeur de retour dans `%eax`
10. Retour en espace utilisateur

- https://syscalls.mebeim.net
- Tracer les appels systèmes : `strace`

Processus et threads

Qu'est-ce qu'un processus ?

- Processus : instance d'un programme qui s'exécute
  - Programme : statique, fichier sur un disque
- L'OS exécute plusieurs processus *simultanément*
- Plusieurs processus peuvent exécuter le même programme
- Non-exemples : plusieurs fenêtres Firefox, plusieurs fenêtres d'un émulateur de terminal, ... $\Rightarrow$ c'est le même processus

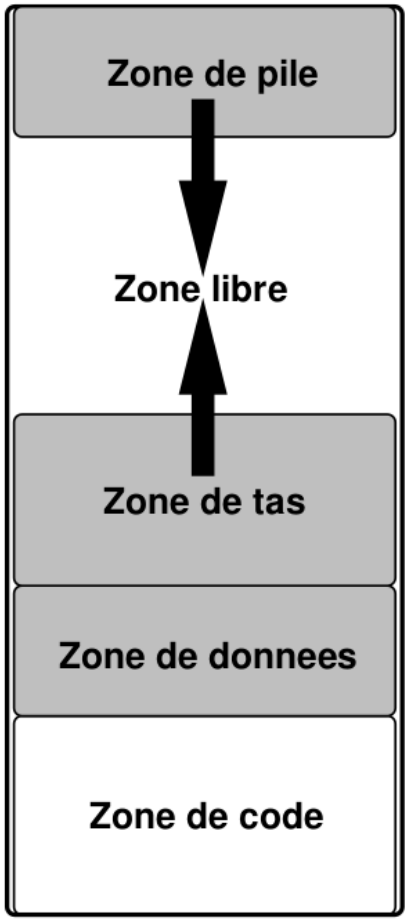
(en français : « processus » ; en anglais : « process »)

Pourquoi des processus ?

- Permet le multi-tâches
- Unité de protection et d'allocation de ressources
  - L'ordonnancement, les droits sur les ressources, ... sont gérés au niveau du processus
- Chaque processus a l'impression d'être seul sur la machine
- Permet le partage de ressources entre applications

Que voit un processus ?

- La (sa) mémoire
  - **Pile** (*stack*)
    Variables locales des fonctions, empilement des appels de fonctions
  - **Tas** (*heap*)
    Mémoire allouée avec `malloc`
  - **Données**
    Variables globales et/ou statiques
  - **Code** (lecture seule)
    Instructions du programmes, chaînes de caractères constantes, ...
- Sa table de descripteurs de fichiers
- Le cœur du processeur sur lequel il s'exécute, son contexte d'exécution

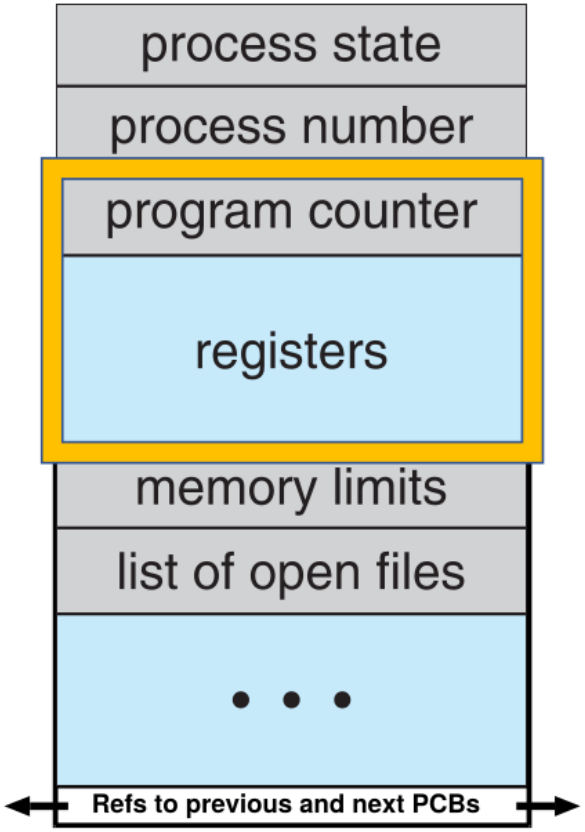


Comment le noyau voit un processus ?

PCB : *Process Control Block*

Structure associée à chaque processus :

- Identifiant : PID
- État (en exécution, prêt, bloqué, mort, ...)
- Priorité
- Adresses mémoires
- *Program Counter*
- Table des descripteurs de fichiers
- Statistiques (temps d'exécution, temps d'attente, ...)
- Utilisateur qui a lancé le processus



Généalogie des processus

- Chaque processus a un processus parent (`getppid()`)
- Chaque processus peut avoir des processus enfants
  - Le parent peut attendre / être notifié de la mort de ses enfants
  - Et récupérer le code de retour
- Cas des orphelins :
  - Un parent par défaut : l'init avec PID = 1

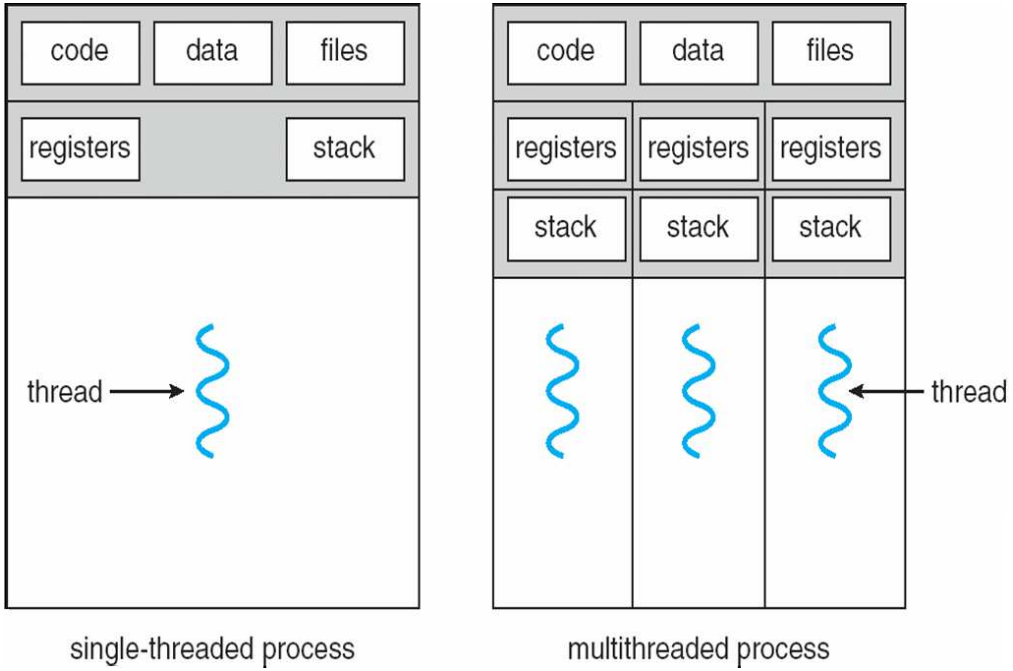
- Voir l'arbre généalogique :
  - `pstree`
  - touche V dans `top`
  - touche F5 dans `htop`

Création d'un processus

1. Appel-système `fork()`
   - Duplique le processus qui appelle `fork()`
   - Le processus appelant `fork()` devient le parent du nouveau processus
   - Duplication de l'espace d'adressage
     - Les zones mémoire des deux processus sont maintenant distinctes !
   - Les deux processus sont identiques (mémoire, contexte d'exécution, ...), sauf pour :
     - Le (P)PID
     - La valeur de retour de `fork()`
2. Éventuellement : appel-système `exec()`
   - Remplace le contexte actuel du processus par l'exécution du programme passé en paramètre

Threads

- *Processus léger*, *fil d'éxécution*, ...
- Fil d'exécution distinct dans un même processus
- Contexte d'exécution de code utilisateur
- Unité de traitement / tâche ordonnancée par le noyau
- Les threads d'un même processus partagent tout, sauf leur pile
- Création avec, au choix :
  - `pthread_create()`
  - Appel système `clone()` avec plus d'options possibles
  - `thrd_create()` depuis C11 (tentative d'avoir une interface portable)



Processus *vs* threads

- Les threads d'un même processus partagent mémoire, fichiers ouverts, signaux, ...
- Un processus avec plusieurs threads peut s'exécuter vraiment sur plusieurs cœurs
  - Un thread par cœur
- Recouvrement des entrées/sorties
  - Exécution d'un autre thread pendant qu'un thread est bloqué dans un appel système
- Le processus devient un conteneur de threads
  - À la création du processus : processus = thread
- Création de thread moins coûteuse que la création d'un processus
- Communication entre threads moins coûteuse, plus simple
- Dans Linux : modèle 1-on-1 : à un thread noyau correspond un thread utilisateur
  - Aussi coûteux qu'un processus
  - Processus multi-threadés avantagés par l'ordonnanceur

Fin d'un thread

- `pthread_exit()` est appelé implicitement ou explicitement
- Attente possible avec `pthread_join()`
- Le dernier thread termine aussi le processus en appelant `exit()`

Fin d'un processus

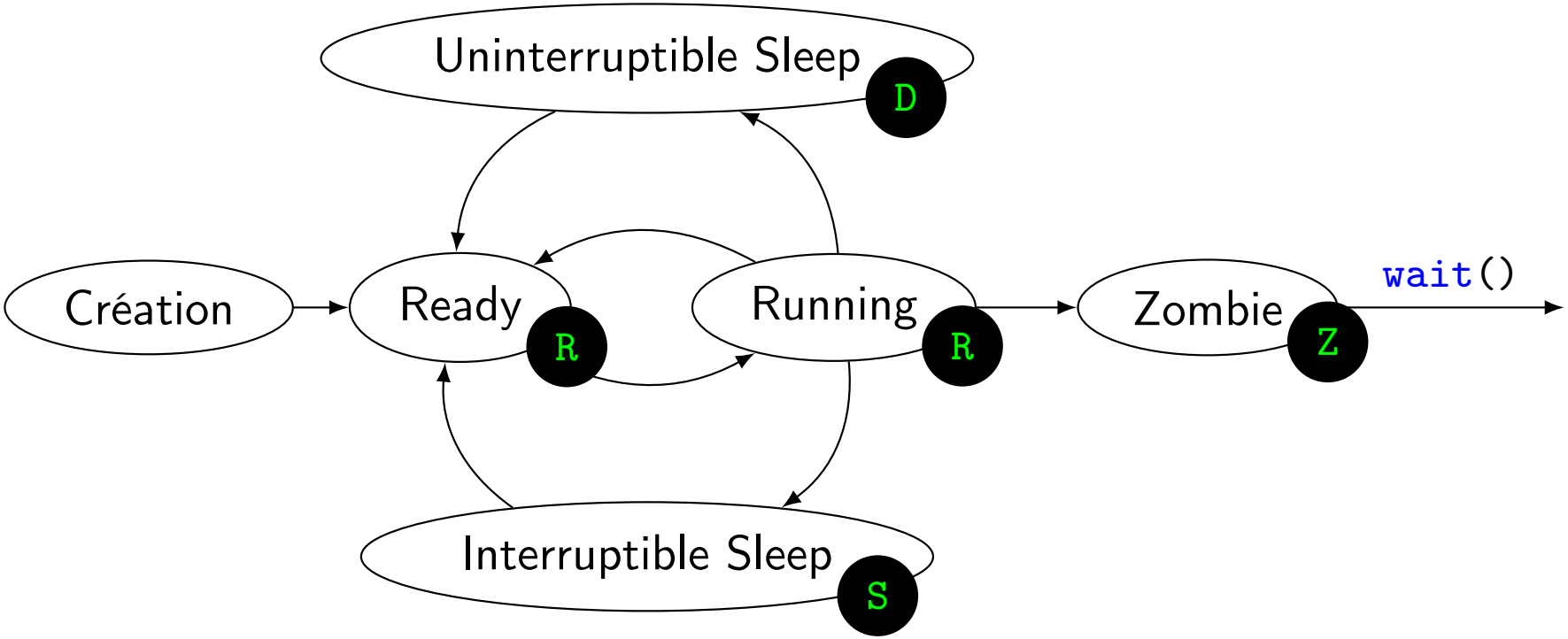
- Code de retour : `return` de la fonction `main()`
- Le processus parent peut attendre la fin d'un processus enfant :
  - `wait()` ou `waitpid()`
  - Recevoir le signal `SIGCHLD`
  - Récupère le code de retour
- Processus dans l'état *Zombie* tant que le parent ne l'attend pas

Exemple en Bash :

```
mon_pid=$$

un_programme_ou_fonction &
pid_task=$!

# faire autre chose...

wait $pid_task
ret_task=$?
```

États d'un processus

- Alternance entre *Ready* et *Running* selon la décision de l'ordonnanceur
- Lors d'un appel système bloquant :
  - *Sleeping* dans une file à part en attendant un événement
  - Puis retour à *Ready* après l'événement
- *Zombie* à la fin du processus tant que le parent ne l'a pas attendu



Comment faire attendre un processus ?

Attente active

- Boucle infinie tant que l'événement n'arrive pas
- Le cœur est occupé
- Très bonne réactivité
- Gaspillage de cycles processeur si trop long

Attente semi-active

- Attente active, mais en donnant l'opportunité d'exécuter un autre processus à chaque fois (*yield*)
- Réactivité imprévisible : on ne sait pas quand le processus sera à nouveau exécuté
- Le processus peut être à nouveau exécuté alors que l'événement n'est toujours pas là

Attente passive

- Demande d'être réveillé quand l'événement est arrivé
- Processus sorti de la liste des processus prêts à être exécutés
- Lors de l'événement : celui qui reçoit l'événement place le processus dans la liste des processus prêts à être exécutés

Interactions entre processus

- Environnement créé avant l'appel à `exec()` :
  - Descripteurs de fichiers ouverts
  - Variables d'environnement
  - ...
- Fichiers :
  - Vim écrit un fichier, gcc le lit et le compile
- Communications inter-processus (IPC) :
  - Tube créé avec `pipe()`
  - Tube nommé (fichier) avec `mkfifo()`
  - Mémoire partagée (SHM, `mmap()`, ...)
- Signaux
- Sockets

Ordonnancement

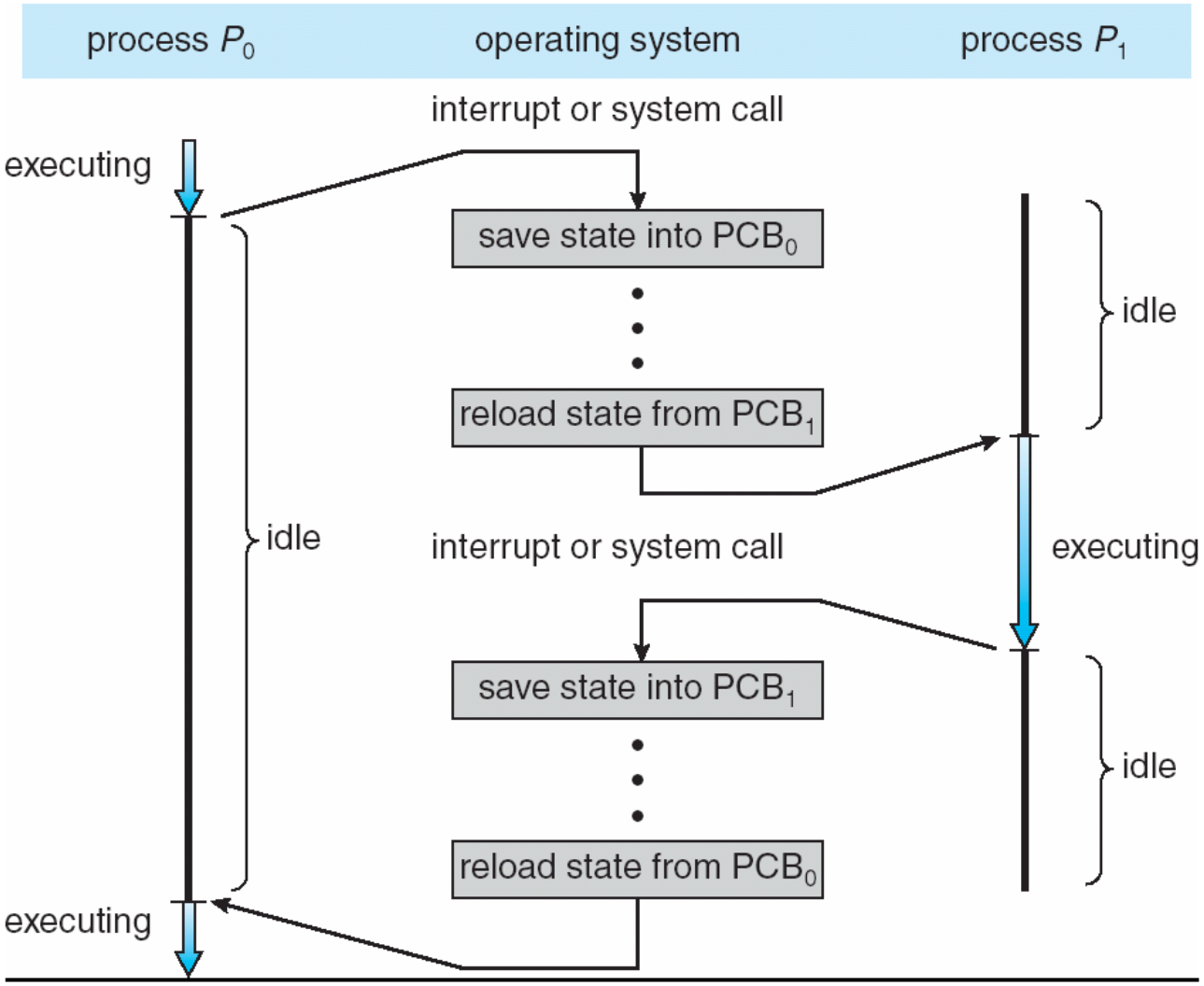
Un ensemble de tâches

- Multiples *tâches* à exécuter simultanément :
  - Processus
  - Threads
  - Tâches du noyau
- Comment passer d'une tâche à l'autre ?
- Quand passer à une autre tâche ?
- Comment choisir la prochaine tâche à exécuter ?

Passer d'une tâche à l'autre

- *Changement de contexte*, *context switch*
- Sauvegarder le contexte d'exécution d'une tâche puis restaurer le contexte d'exécution d'une autre tâche
- Coût non-négligeable
  - Sauvegarde/restauration du pointeur d'instruction et registres
  - Sauvegarde/restauration des registres flottants
  - Changement de la table de traduction d'adresses
  - Entraîne des défauts de cache

Changement de contexte



Ordonnancement

- *Scheduling*
- Définir un ordre d'exécution, éventuellement avec des limites de durée
- Tout un domaine, pas spécifique aux systèmes d'exploitation

Ordonnanceur

- Entité responsable de l'ordonnancement, prend les décisions
- Composant exécuté par le noyau
- Le noyau en tant qu'entité n'est pas à ordonnancer
  - Mais le noyau a des tâches internes à ordonnancer
- Comment invoquer l'ordonnanceur ?

Ordonnancement (pseudo-)coopératif

- Ordonnancement coopératif : explicite par les applications
  - `sched_yield()` pour passer la main
  - `sleep()` pour attendre
- Ordonnancement pseudo-coopératif : implicite
  - Profiter de chaque appel système

- Aucune garantie d'ordonnancement régulier
- Dépend du bon vouloir des applications
- Problème d'équité et de réactivité
  - On ne sait pas quand on récupérera la main
- Coopération nécessaire entre les tâches

Préemption

- Désordonnancer une tâche de force
- Si une tâche s'exécute depuis trop longtemps
- Si une autre tâche est plus prioritaire

- Léger surcoût : on vérifie de temps en temps s'il faut désordonnancer de force
- Gain énorme en équité et réactivité !

- Tranche de temps d'exécution donnée à un processus : *timeslice*

Comment / quand préempter ?

- C'est le noyau qui préempte
- Il faut que le code du noyau pour préempter soit exécuté régulièrement, même si l'application ne fait pas appel au noyau !

⇒ Profiter des interruptions de l'horloge !

- Une interruption exécute du code noyau et interrompt l'exécution d'un processus

Algorithmes d'ordonnancement

Quel est le prochain processus à exécuter ?

Objectifs

- Optimiser l'utilisation des cœurs
  - Ne pas gaspiller des cycles à ne rien faire
- Maximiser la satisfaction de l'utilisateur
  - Interaction fluide par exemple
- Réactivité
  - Ne pas trop attendre pour exécuter un processus
- Équité
  - Tous les processus doivent s'exécuter à un moment
- L'ordonnanceur doit s'exécuter rapidement

Comment stocker les processus ?

- Objectif : chercher le prochain processus à exécuter doit être rapide, manipulation de l'ensemble des processus (ajout, suppression, déplacement, ...) doit être en temps constant
- Donc impossible de parcourir un tableau / une liste jusqu'à trouver un processus prêt à s'exécuter
- Le nombre de processus varie : certains disparaissent, d'autres sont créés

  ⇒ A priori une liste plutôt qu'un tableau

- L'ordonnanceur a besoin de connaître rapidement le prochain processus **prêt** à s'exécuter

⇒ Stockage uniquement des processus prêts dans une liste dédiée : la *runqueue*

- Les autres processus attendent dans une liste dédiée à l'événement attendu

Ordonnanceur FIFO

- Utilisation d'une file *First In First Out*
- Processus à exécuter en tête de file
- Sorti de la file quand le processus a fini ou attend (pas de préemption)

- Avantages :
  - ? ?
  - ? ?
- Inconvénients :
  - ? ?

Ordonnanceur FIFO

- Utilisation d'une file *First In First Out*
- Processus à exécuter en tête de file
- Sorti de la file quand le processus a fini ou attend (pas de préemption)

- Avantages :
  - Très léger surcoût (*overhead*)
  - Choix de la prochaine tâche à exécuter en $\mathcal{O}(1)$
- Inconvénients :
  - Famine

Ordonnanceur Round-Robin

- *Round-Robin* : tourniquet
- FIFO avec préemption
- Chaque processus est exécuté chacun son tour
- À chaque préemption, on exécute le prochain processus

- Avantages :
  - ? ?
  - ? ?
- Inconvénients :
  - ? ?

Ordonnanceur Round-Robin

- *Round-Robin* : tourniquet
- FIFO avec préemption
- Chaque processus est exécuté chacun son tour
- À chaque préemption, on exécute le prochain processus

- Avantages :
  - Pas de famine
  - Choix de la prochaine tâche à exécuter en $\mathcal{O}(1)$
- Inconvénients :
  - Pas de gestion des priorités

Autres idées d'ordonnanceurs

- Exécution de la tâche la plus courte ou qui finira le plus rapidement
  - Toujours juste ?
  - Comment prévoir la durée d'exécution ?
- Mécanisme de priorités

Priorités

- Nombre de priorités possibles fixe
- Priorité d'un processus dynamique
  - Ajustée par l'ordonnanceur, selon l'activité du processus dans ses précédentes timeslices
- Une part de la priorité est ajustable par l'utilisateur
  - `nice`
  - Intervalle de -20 à 19
  - Petite valeur ⇔ grande priorité

Comment exploiter les priorités ?

- Exécuter les tâches prioritaires d'abord ?
  - Utile ?

- Exécuter les prioritaires plus souvent
- Exécuter les prioritaires plus longtemps
  - Adapter la timeslice selon la priorité

Stocker les processus selon leur priorité

- *Runqueue* triée par priorité ?
  - Conserver le tri nécessite un algorithme linéaire en le nombre de processus
  - Chaque opération d'ordonnancement coûtera cher
  - Risque de famine !

- Une liste de processus par priorité
- Au sein d'une liste, tous les processus ont la même priorité
  - Manipulation en temps constant
- Nombre fixe de priorités disponibles $\Rightarrow$ nombre fixe de listes
- Les listes sont parcourues plus ou moins souvent selon la priorité

Ordonnanceur temps-réel

- Jusqu'à maintenant : pas de garantie sur *quand* un processus s'exécutera la prochaine fois
- Certains cas peuvent nécessiter une telle garantie (embarqué, robotique, santé, audio, vidéo, ...)
- On souhaite une garantie qu'un événement sera traité avant un délai fixé ⇒ *temps-réel* (RT : *real-time*)

- Par défaut, Linux n'est pas temps réel
- Pendant de nombreuses années : un patch à appliquer au noyau
  - Patch fusionné dans Linux depuis 6.12
- Rend préemptible des sections critiques du noyau et ajoute un ordonnanceur temps-réel (complexe)
- Nécessite d'indiquer qu'on souhaite utiliser l'ordonnanceur temps-réel

Exécution *simultanée* ?

Si on a un unique cœur, a-t-on vraiment une exécution simultanée de plusieurs processus ?

Exécution *simultanée* ?

Si on a un unique cœur, a-t-on vraiment une exécution simultanée de plusieurs processus ?

- Non, uniquement une alternance d'exécution de processus
- Mais c'est transparent pour les processus

Et les machines multi-cœurs ?

- Plusieurs cœurs ⇒ plusieurs flux d'exécutions vraiment simultanés
  - Un flux par cœur
- Toujours de la préemption (il y a généralement plus de tâches que de cœurs)
- Nouvelle question : sur quel cœur exécuter quelle tâche ?
  - Notion de localité / affinité : proche de la mémoire / du matériel manipulé par le processus
  - Changer de cœur pendant l'exécution d'un processus ⇒ défauts de cache
  - Casser l'affinité parce qu'un cœur est libre ?

Structures de données pour machines multicœurs

- Le cœur sur lequel s'exécuter introduit une dimension supplémentaire à la runqueue

Une seule runqueue

- *Load Sharing*
- Tous les cœurs y piochent
- Contention pour y accéder si beaucoup de cœurs
- Simple à implémenter à utiliser
- Comment garder une trace des affinités ?

Une runqueue par cœur

- *Load Balancing*
- Chaque cœur a sa propre runqueue
- Pas de contention
- Comment / quand équilibrer les différentes runqueues ?

Mémoire

Rappel : différents types de mémoire

Mémoire volatile

- Qui s'efface en l'absence de courant
- Plus ou moins rapide
- Vue dans cette section

- Registres des cœurs
- Caches des cœurs
- RAM

Mémoire non-volatile

- Stockage persistent (combien de temps ?)
- Relativement lent
- Vue dans la section sur les I/O

- Disques durs (mécanique, SSD, NVMe)
- Disquettes, CD, DVD, Blu-Ray, ...
- Bande (oui, oui !)

Cache *vs* bufferisation

Cache

- Zone de stockage intermédiaire
- Plus rapide, mais plus petite
- Données récemment accédées pour y accéder à nouveau, préchargement de données qui pourraient être bientôt chargées
- Déplacement vers une mémoire plus lente lorsqu'il n'y a plus de place (*évincement*)
  - Quelles données évincer ?
- Exemples :
  - Dans le contrôleur de disque : matériel
  - Dans le processeur : surtout matériel (caches L1/L2/L3)
  - Cache des pages mémoires, *swap* : logiciel

Bufferisation

- Stockage des données dans une zone temporaire
- Traitement des données lorsqu'il y en assez
- Exemples ?

Encore une ressource à partager

- Chaque processus a besoin de mémoire pour être exécuté
- Plusieurs processus existent simultanément $\Rightarrow$ leur mémoire aussi

Comment répartir la mémoire entre processus ?

Une solution : le partitionnement

Avec n processus, on découpe la mémoire en n partitions de même taille

Inconvénients

- Au maximum n processus
- Chaque processus obtient la même quantité de mémoire
  - Un processus qui n'utilise pas tout : gaspillage
  - Un processus qui a besoin de plus : famine
- Chaque processus a un intervalle d'adresses mémoire qu'il peut utiliser
  - Que se passe-t-il si un processus lit/écrit une adresse en-dehors de sa partition ?

Un partitionnement avec moins d'inconvénients

On peut imaginer :

- Des partitions de tailles différentes
  - Comment décider quelle taille attribuer à un processus ?
- Un nombre de partitions qui évolue dans le temps
  - Au fur et à mesure que de nouveaux processus sont créés
  - Mais quid d'une évolution de la consommation mémoire d'*un* processus ?
- Un support matériel pour empêcher les processus de manipuler la mémoire hors de leur partition
  - Mais parfois on veut partager de la mémoire entre processus

Pagination

- Pas de gros blocs mémoire par processus
- Mais de petites *pages* allouées aux processus au fur et à mesure des besoins
  - Généralement 4 Ko, mais dépend de l'architecture

Avantages

- On peut gérer la mémoire à faible granularité
- Pas (peu) de gaspillage mémoire

Inconvénients

- Support matériel bien plus complexe
- Pas de garantie que les pages d'un même processus soient contiguës en mémoire
  - Problème : un objet à cheval sur deux pages n'a plus des adresses continues !

Et pourtant c'est ce qui est utilisé !

Quels besoins mémoire ont les processus ?

- Avoir un espace d'adressage contigu (au moins par gros morceaux)
- Allocation de plus ou moins de mémoire au cours du temps
  - On ne peut pas prévoir la consommation mémoire d'un processus
  - La consommation mémoire peut dépendre des données d'entrée
- Avoir assez de mémoire
- Avoir toute la mémoire de la machine
- Pouvoir parfois partager des espaces mémoires entre différents processus
- Mais par défaut empêcher qu'un processus vienne lire/écrire dans une mémoire qui ne lui appartient pas

Autres caractéristiques

- Les processus n'utilisent pas forcément toute la mémoire qu'ils allouent
  - Jamais utilisée ou juste pas simultanément
- Plusieurs processus peuvent avoir besoin de charger dans leur mémoire les mêmes bibliothèques / fichiers
  - Tous les processus ont besoin de la `libc`
  - Gimp, Gedit, Evince, ... ont tous besoin de la bibliothèque GTK
  - Bibliothèque chargée en *lecture seule*

Mémoire virtuelle

- Solution pour répondre aux besoins tout en utilisant la pagination : **virtualiser la mémoire**
- Toujours une unique mémoire physique
- Mais deux types d'adresses

Adresse virtuelle

- Adresse manipulée par les processus, le noyau et même le processeur
- Dès qu'on affiche une adresse (`printf("%p", &var);`), c'est une adresse virtuelle
- Chaque processus peut utiliser tout l'espace d'adresses virtuelles possible, qui lui est propre
  - Une adresse virtuelle appartenant à un processus n'a pas de sens pour un autre processus

Adresse physique

- Adresse manipulée au dernier moment pour accéder à la mémoire
- Il faut faire la traduction adresse virtuelle $\rightarrow$ adresse physique
  - C'est le matériel qui s'en charge (*cf* plus loin)

Des adresses vers quoi ?

- Une adresse désigne un octet
- Les processeurs manipulent des octets
  - Parfois par paquets de 1, 2, 4, 8, ... octets
  - Avec parfois des contraintes d'*alignement*
    - Quelle est la taille d'une structure en C ?
    - Contraintes de certaines fonctions
- Comment accéder à un bit précis d'un octet ?
  1. Chargement de l'octet entier dans un registre (fait de façon implicite par le processeur)
  2. Utilisation d'instructions bit-à-bit (&, |, ~, <<, >>, ...)

Où sont les octets ?

- Cache processeur, RAM, disque, ... peu importe
- Avantage de la mémoire virtuelle : indépendant du vrai stockage physique
- Sera géré au dernier moment à l'aide du matériel

Intérêts de la mémoire virtuelle

- Abstraction du stockage physique
  - Au pire, on peut stocker sur le disque
- Le code et les données non utilisées peuvent rester sur le disque
  - Chargement au dernier moment
- Véritable allocation des pages au dernier moment
  - On peut allouer plus de mémoire que disponible physiquement !
- Chaque processus a l'espace d'adressage virtuel complet pour lui tout seul
  - Peut se croire tout seul à utiliser toute la mémoire
- Isolation de la mémoire de chaque processus
- Abstraction pour la programmation : on manipule des adresses linéaires, quelque soit le vrai stockage physique
- Mutualisation facile entre processus :
  - Plusieurs adresses virtuelles pointent vers la même adresse physique
- Droits d'accès possible à la granularité de la page

Inconvénients de la mémoire virtuelle

- Complexité
  - Besoin de l'aide du matériel
  - Mais le noyau gère quand même une bonne partie
- Performances cruciales
  - Traduction nécessaire à chaque accès mémoire !
- Consommation mémoire
  - Il faut stocker les tables de traduction

Traduire une adresse virtuelle en adresse physique

Une adresse virtuelle est composée de :

- Numéro de page mémoire (virtuelle)
  - PFN : *Page Frame Number*
- Décalage au sein de la page

Le numéro de page mémoire correspond à un indice dans une **table de page** :

- On récupère une PTE : *Page Table Entry*
- Contient (entre autres) l'adresse physique du début de la page mémoire physique

On peut ensuite appliquer le même décalage sur l'adresse physique.

Les *Page Table Entries*

- Chaque processus possède sa table de pages
- La table de page est composée de PTE

Chaque PTE contient :

- L'adresse physique de la page
- Bits de validité
  - Est-ce que la page physique existe vraiment ?
- Bits de protection : `RO`, `RW`, `X`, ...
- Bits d'état (modifié / `dirty`, ...)
- Bits pour des statistiques

La structure de la PTE dépend de l'architecture et de l'OS

Exemple de traduction d'adresse

- Hypothèses :
  - Adresse virtuelle sur 32 bits
  - Adresse physique sur 24 bits
  - Pages de 4 Ko (= 4096 octets)
- Page de 4096 octets ⇒ besoin de 12 bits pour décrire le décalage au sein de la page
- Il reste 12 bits pour décrire la page physique
  - On ne pourra utiliser au maximum que 2^{24} octets = 16 Mo de mémoire physique
- Pour l'adresse virtuelle :
  - Toujours 12 bits de décalage au sein de la page
  - Avec les 20 bits restants :
    - Faire croire qu'on peut gérer 2^{32} octets = 4 Go de mémoire
    - ⇒ Besoin de 2^{20} PTE par processus
    - Ne pas utiliser tous les 20 bits ?
    - ⇒ Mais le processeur aime bien les puissances de 2

Stockage des tables de pages : cahier des charges

- On veut associer à chaque numéro de page une PTE
- La traduction doit être la plus rapide possible
- Structure de données et manipulation suffisamment simples pour être manipulable directement par du matériel dédié
- Structure de données adpatée aux accès concurrents
- Propriété sans doute à exploiter : une grande partie de l'espace d'adressage n'est pas utilisé

Structures de données pour les tables de pages

Le plus simple : un tableau de PTE

- La table doit être entièrement allouée
  - Pour pouvoir tester si la page est valide ou non
- Problème : la table de pages prend de la place en mémoire
  - Si 2^{20} PTE de 4 octets = 4 Mo pour stocker toute la table

Table pas entièrement allouée

- Liste d'intervalles de mémoire valide
  - Mauvaise complexité du parcours de liste
- Arbre binaire d'intervalles valides
  - Support matériel difficile
  - Modifications pas atomiques ?
- Table hachée (IA64, PowerPC)
  - Fonction de hachage qui donne l'indice dans le tableau à partir du PFN
  - Liste si plusieurs PFN arrivent dans la même case
  - Temps de recherche pas constant
  - Optimal en consommation mémoire
  - Complexe pour avoir un support matériel

Autre structure : table de pages à plusieurs niveaux

- Un tableau de PTE $\Leftrightarrow$ un seul niveau
- Table à plusieurs niveaux : un tableau d'indirection vers des sous-tableaux
  - Un tableau de niveau 1 avec N cases
  - N tableaux de niveau 2 avec M cases
  - etc
- Intérêt : permet de supprimer les tableaux de niveau > 1 pour les zones mémoires invalides
- Seul le tableau de niveau 1 est forcément entièrement alloué
- L'adresse virtuelle est découpée en un ensemble d'index dans chaque niveau :
  - Premier bits : entrée dans la table de premier niveau
  - Bits suivants : entrée dans la table du niveau suivant
  - ...
  - Derniers bits : décalage dans la table de page
  - Tous les bits ne sont pas forcément utilisés !

Cas extrêmes des tables de pages à plusieurs niveaux

Processus avec une seule page mémoire

- Il n'y a qu'une table allouée par niveau
- Très petit comparé à une table linéaire complète

Processus qui utilise *toute* la mémoire virtuelle

- Le dernier niveau fait la même taille qu'une table linéaire complète
- Et il y a les tables de niveaux inférieurs en plus
- Mais c'est le pire cas, extrêmement rare !

Table de page à 4 niveaux dans Linux x86_64

- 4 niveaux utilisés
  - Les nouvelles architectures peuvent en supporter 5 (flag `la57` dans `/proc/cpuinfo`)
- Pointeurs de 8 octets (64 bits)
- Mais seulement 48 bits d'adresse virtuelle réellement utilisés (entrée `address size` dans `/proc/cpuinfo`) :
  - 9 bits de PGD (*Page Global Directory*)
  - 9 bits de PUD (*Page Upper Directory*)
  - 9 bits de PMD (*Page Middle Directory*)
  - 9 bits de PTE (*Page Table Entry*)
  - 12 bits de décalage dans la page

$\Rightarrow$ Permet d'adresser 256 To de mémoire virtuelle !

Table de page à 4 niveaux dans Linux x86_64

Pourquoi **9** bits par niveau ?

- Pour faciliter la manipulation : chaque (sous-)table doit tenir exactement dans une page
- Une page = 4096 octets
- Une PTE = 8 octets
  - 4096 / 8 = 512 PTE dans une page
  - $512 = 2^9$ donc besoin de 9 bits pour savoir où est le PTE dans la page
- Un pointeur (une adresse) = 8 octets
  - Mêmes calculs : 512 pointeurs par page donc 9 bits

Pagination fainéante

- Allouer de la mémoire physique prend du temps

$\Rightarrow$ Dire que l'allocation a réussi, créer la PTE, mais allouer la mémoire physique au dernier moment

  - Lors de l'accès à la mémoire

- Ne pas dupliquer une page tant qu'elle n'est pas modifiée
- Charger une page depuis le disque que lors de l'accès
- Accès à une page allouée de façon fainéante : **défaut de page**

Défaut de page

- Accès à de la mémoire physique pas encore allouée
- On demande au matériel de traduire une adresse virtuelle qui n'a pas d'adresse physique
- Le processeur ne sait pas quoi faire
  - Il demande à l'OS en envoyant une exception
- Plusieurs cas possibles :
  - Mémoire pas allouée physiquement : allocation
  - Page mémoire sur le disque : chargement en RAM
  - Accès en écriture à une page dupliquée : duplication (*copy-on-write*)
  - Adresse invalide / accès interdit : *segmentation fault*

Projections

- Projection d'une zone mémoire dans l'espace mémoire d'un processus
  - Depuis le disque
  - Depuis une autre zone mémoire
- Appel système `mmap()` :
  - Projection publique (`MAP_SHARED`) : modifications effectuées répercutées dans le fichier
  - Projection privée (`MAP_PRIVATE`) : modifications propres au processus, perdues à la fin du processus
  - Projection anonyme (`MAP_ANONYMOUS`) : pas de fichier de support, utilisation de `/dev/zero`
- Utilisation pour la pile, le tas, le code, les bibliothèques, ...
- Permet de mutualiser les zones mémoires communes (bibliothèques, ...)

Fork et *copy-on-write*

- Les copies mémoires coûtent cher
- Un `fork()` pour créer un processus duplique tout l'espace mémoire
- Mais pas vraiment :
  - Retarder au maximum les duplications
  - Les pages des deux processus pointent vers la même page physique
    - Économise également la mémoire physique
  - Les deux pages sont mises en lecture seule
  - Lors d'une modification par un des processus ⇒ défaut de page ⇒ copie de la page physique

Swap

- « Espace d'échange »
- Extension de la mémoire physique RAM
- Généralement stocké sur un disque dur ⇒ lent !
  - Partition dédiée
  - Fichier
  - Ou même en RAM (avec compression : *cf* zRAM)
- Si besoin de libérer de la mémoire en RAM : déplacement de pages dans le swap
- La PTE contient l'information que la page physique est dans le swap
  - Rechargement en RAM si besoin d'y accéder
- Seules les pages privées et/ou anonyme vont dans le swap
  - Pour les projections publiques (de fichier) : sauvegarde directement dans le fichier
- Entièrement géré par l'OS
  - Quand swapper, quelles pages swapper, ...

Support matériel

MMU : *Memory Management Unit*

- Circuit dédié du processeur pour traduire les adresses
- Sait lire la table de page, gère les droits d'accès, déclenche les exceptions pour défaut de page
  - Sur x86 : l'adresse du PGD est mise dans le registre `cr3`
- Problème : reste trop lent
  - Nécessaire pour chaque accès mémoire
  - La traduction d'*une* adresse demande *plusieurs* accès mémoire !
- Certaines architectures n'en ont pas
  - Exemple : MIPS
  - Traduction logicielle par des exceptions

TLB : *Translation Lookaside Buffer*

- Technique habituelle : lorsque c'est trop lent, on met un cache
- TLB : cache des dernières traductions dans la MMU
- Très rapide, mais petit : quelques centaines d'entrées
- TLB miss $\rightarrow$ traduction par la MMU / l'OS

Changement de contexte

- Chaque processus a sa table des pages
- Changement de contexte $\Rightarrow$ changement de processus $\Rightarrow$ changement de table de pages
- Changement de l'adresse dans le registre `cr3` sur x86
- Vider le TLB

Et la mémoire du noyau ?

- Espace virtuel spécifique au noyau
  - Accessible uniquement en mode privilégié
  - Pas modifié lors de changements de contexte
- Le noyau est un contexte d'exécution
  - A un espace virtuel qui lui est propre
  - Et donc une table de pages
- Le noyau peut accéder à toutes la mémoire de tous les processus
- Le noyau a besoin de :
  - Son code : binaire autosuffisant
    - Pas besoin de bibliothèques
    - Fonctions de base réimplémentées pour ne pas dépendre de la `libc`
  - Ses propres données
  - Tout petit : 10-100 Mo !

Espace virtuel dans Linux 64 bits

- Divisé en deux espaces :
  - Mémoire pour le noyau
  - Mémoire du processus courant
  - 128 To virtuels chacun
- L'espace noyau est partagé entre tous les processus
  - Mais une seule fois en mémoire physique

Que faire si toute la mémoire physique est utilisée ?

- Toute la RAM et tout le swap sont complets
- Comment libérer de la mémoire ?
- Tuer un processus
  - Fait par le *OOM-killer*
  - OOM : *Out Of Memory*
- Quel processus tuer ?
  - Celui qui demande de la mémoire ?
    - Peut-être qu'il n'avait encore rien demandé...
  - Un processus qui va libérer la quantité de mémoire dont on a besoin
  - Un processus pour lequel on ne va pas perdre beaucoup de données
  - Pas un processus nécessaire au fonctionnement du système (pas l'init, pas le serveur graphique, etc)
- Linux maintient un *OOM-score* pour chaque processus
  - *cf* `man proc_pid_oom_score_adj`

Machines NUMA

- *Non-Uniform Memory Access*
- Certaines zones mémoires physiques sont plus proches d'un processeur que d'un autre
  - Surtout dans les machines avec plusieurs processeurs
  - Possible même au sein d'un unique processeur (*sub-numa clustering*)
- On parle de *banc NUMA* ou *nœud NUMA*
- Chaque processeur peut toujours accèder à toute la mémoire physique
  - Mais accéder à la mémoire hors de son banc NUMA est plus lent
- Amène plusieurs questions :
  - Sur quel banc NUMA allouer la mémoire ?
  - Manque de place sur un banc NUMA : allouer sur un autre banc NUMA ou déplacer d'autres pages ? Plutôt faire exécuter le processus par un autre processeur ? Swapper ?
- Par défaut : allocation *first touch*
  - Dans les applications multithreadées, la page est allouée physiquement dans le banc NUMA du cœur qui y accède pour la première fois

Guider la gestion de la mémoire

On peut donner des indications au noyau :

- Adresse virtuelle et permissions souhaitées avec `mmap()`
- Changer les permissions d'une page mémoire : `mprotect()`
- Empêcher une page d'être swappée : `mlock()`
- Indiquer dans quel nœud NUMA allouer la mémoire : `man 3 numa` ou utiliser Hwloc
- Comment la page mémoire sera utilisée : `madvise()`

Métriques de la consommation mémoire

Par processus

- `cat /proc/$pid/maps`
- Différentes colonnes dans `top` et `htop`
  - *cf* `man top` (excellent) et `man htop`
  - `VIRT` : toute la mémoire virtuelle utilisée par le processus
  - `SHR` (*shared*) : mémoire partagée entre processus
  - `RES` (résiduelle) : la mémoire physique utilisée par le processus (inclut `SHR`)
  - `MEM%` : `RES` / quantité de RAM

Vue globale du système

- `free -h`
  - Quantité de RAM totale
  - Mémoire partagée entre processus
  - Buffers noyau et cache I/O
  - Disponible à l'utilisation (exclut le cache I/O)
  - Utilisée : quantité de RAM totale – disponible
  - Libre : mémoire pas utilisée
- Linux utilise la mémoire libre comme cache I/O

Résumons !

Stockage

I/O

- *Input/Output*
- Entrées/sorties de données dans l'ensemble {processeur, RAM}
- Peut désigner plusieurs choses :
  - Tous les périphériques : cartes PCI, USB, clavier, souris, réseau, écran, son, console série, ...
  - La mémoire non-volatile : **disques**

Problématiques du stockage de fichiers

- Intégrité des données
- Performance
- Comment stocker les données ?
- Et les métadonnées ?
  - Attributs, taille, droits, ...
- Comment récupérer les données après un crash ?
- Comment gérer les accès concurrents de différents processus ?

Solution : utiliser des sytèmes de fichiers

Systèmes de fichiers

- Définit comment sont stockées les (méta-)données sur le disque
- Permet de distinguer le stockage physique (ensemble d'octets) des représentations logiques (fichiers, répertoires, ...)
- Abstrait le type de support physique

Ne pas utiliser de systèmes de fichiers ?

- On peut ouvrir directement le périphérique disque pour lire et écrire des octets
  - Périphérique « *bloc* »
  - Par exemple : `/dev/sda3`
- Les processus doivent se mettre d'accord sur qui écrit où

Pile d'accès au stockage

1. Interface de haut niveau : `fopen`, `fread`, `fwrite`, ... (bibliothèque C)
2. Appels systèmes : `open`, `read`, `write`, ...
3. Couche de virtualisation des fichiers dans le noyau
   - *Virtual File System* dans Linux
   - Systèmes de fichiers : Ext4, NTFS, ZFS, BTRFS, NFS, SWAP, SSHFS (FUSE), ...
4. Couche de virtualisation des blocs
   - *Block Device Layer* dans Linux
   - Interface d'accès au matériel : IDE, SATA, NVMe, ...
   - Possibilité de RAID, LVM, chiffrement, ...
5. Drivers de chipsets et périphériques
6. Périphériques : stockage physique des blocs de données

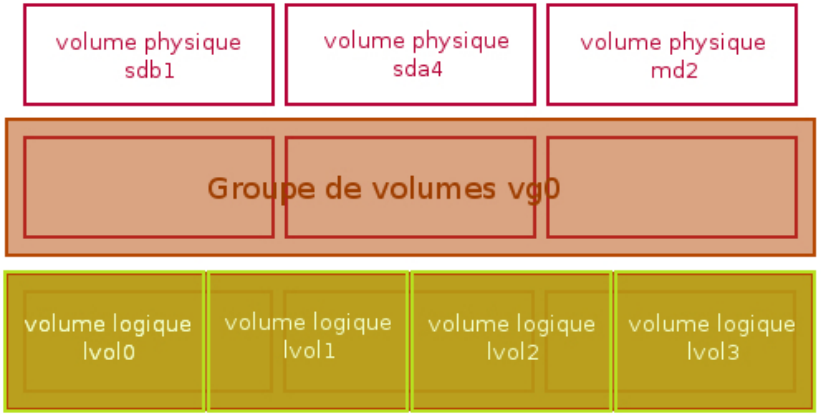
Partitions

- Le disque peut être découpé en zones : *partitions*
- Le découpage est indiqué dans la *table de partitions*
  - Dans les premiers octets du disque
  - Aujourd'hui au format GPT
    - Avant : format MBR
  - Une entrée pour chaque partition :
    - GUID : *Globally Unique IDentifier*
    - Première et dernière adresse
    - Attributs (amorçable, lecture seule, ...)
    - Nom
- Le BIOS et l'OS doivent savoir lire la table de partitions
- Dans une partition :
  - Un système de fichiers
  - ou un agrégateur de disques : LVM

- `lsblk -f`
- `sudo fdisk -l`

LVM : *Logical Volume Manager*

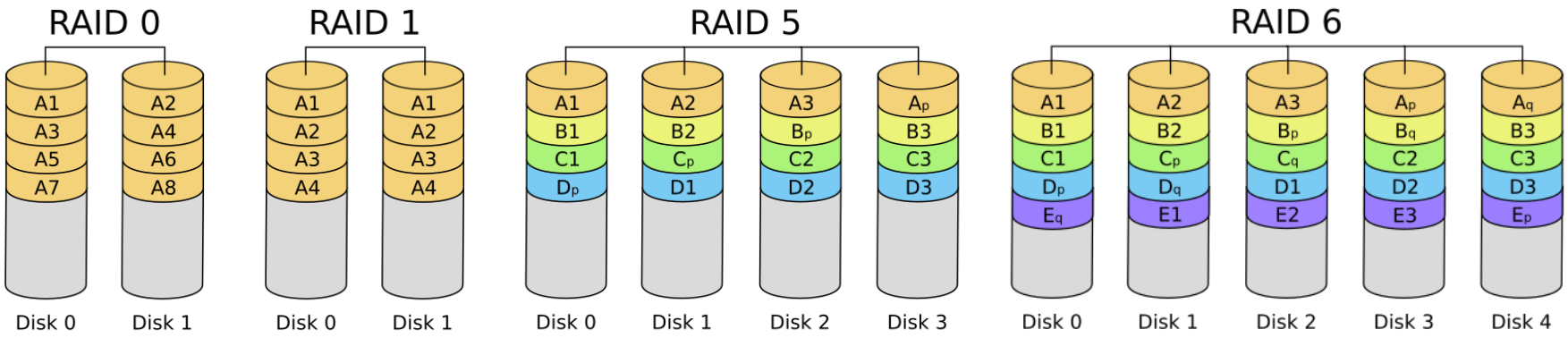
- Abstrait la notion de partitions physiques
- Permet d'avoir des volumes logiques
- Par exemple :
  - Un système de fichiers qui s'étend sur plusieurs disques
  - Opérations de redimenssionement facilitées
  - Instantanés (*snapshots*)
- Gestion beaucoup plus flexible du stockage
- Complétement logiciel !



https://doc.ubuntu-fr.org/lvm

RAID : *Redundant Array of Independent Disks*

- Techniques pour répartir les données sur plusieurs disques
- Objectifs :
  - Performances : parallélisation des accès
  - Redondance des données : permet de perdre ≥ 1 disque(s)
    - Un disque (presque) défaillant ? Il suffit de le changer
- Implémentation :
  - Matérielle : support par la carte mère ou périphérique dédié
  - Logicielle : en-dessous du système de fichiers (`mdadm`), ou directement géré par le système de fichiers (LVM, ZFS, BTRFS, ...)



https://fr.wikipedia.org/wiki/RAID_(informatique)

Structure des systèmes de fichiers

- Lecture du contenu brut d'un système de fichiers : ensemble d'octets dont l'organisation est gérée par le système de fichiers
- Organisation en arbre :
  - Feuilles : fichiers
  - Nœuds intermédiaires : répertoires
- Fichier :
  - Suite d'octets qui n'a pas de sens pour le système de fichiers
  - A du sens uniquement pour l'application qui sait lire/écrire ce fichier
- Répertoire : suite de descripteurs d'enfants
- Liens :
  - Symbolique (`ln -s`) : fichier contenant le chemin cible
  - Physique (`ln`) : rupture de la structure d'arbre, deux branches arrivent sur le même nœud
  - Les liens ne sont pas forcément supportés par tous les systèmes de fichiers !
- I-nœuds (*inode*) : méta-données d'un nœud (taille, droits, type, dates, ...)

Stockage des fichiers sur disque

- Les fichiers sont découpés par blocs / secteurs
- On ne peut pas allouer plus petit sur le disque
- Stockage des différents blocs d'un même fichier pas forcément contigu...
  - Besoin de défragmenter de temps en temps pour minimiser les déplacements de la tête de lecture
  - Optimisations pour éviter la fragmentation
  - Peut être géré par le système de fichiers
- Les fichiers peuvent être à trous :
  - Seuls les blocs utiles sont alloués
  - Écrire à la fin d'un fichier en laissant du vide au début

$\Rightarrow$ Taille du fichier $\neq$ occupation sur le disque

- Jouer avec `dd count=... seek=...` pour l'observer

On retrouve des questions similaires à la gestion de la mémoire !

Accès aux disques

- L'accès aux disques est **lent**
- Plusieurs processus peuvent avoir besoin d'accèder au même disque en même temps
- Les accès physiques sont séquentialisés (si pas de RAID)
  - $\Rightarrow$ Potentielle attente pour que sa requête I/O soit traitée
- Partager l'accès aux disques entre processus $\Rightarrow$ ordonnancement des I/O

On retrouve des questions similaires à l'ordonnancement des processus !

Montage

- Les partitions (et leurs systèmes de fichiers) sont disponibles sous forme de fichiers blocs dans `/dev/`
- Montage : insérer l'arborescence d'un système de fichiers dans une autre arborescence
  - Commande / appel-système `mount`
  - Montages à réaliser dans `/etc/fstab`
- Paramètres :
  - Fichier bloc à monter
  - Dossier qui sera la racine de l'arborescence à insérer
  - Type de système de fichiers
  - Options éventuelles

- `mount` appelé tout seul liste tous les montages actuels

Implémentation d'un système de fichiers

Les systèmes de fichiers font le lien entre le stockage :

- périphériques bloc (disques durs)
- protocole réseau (NFS, SSHFS, ...)
- des données générées à la demande

et le Virtual File System :

- Même interface implémentée par tous les systèmes de fichiers :
  - Ouvrir / fermer un fichier
  - Lire / écrire dans un fichier
  - Lister le contenu d'un dossier
  - Obtenir les métadonnées d'un fichier / dossier
  - *etc*

Systèmes de fichiers particuliers

- Systèmes de fichiers virtuels :
  - Sans stockage physique associé
  - `proc` : permet d'obtenir des informations sur les processus (`/proc`)
  - `sys` : permet d'exposer des objets du noyau, informations sur le matériel notamment (`/sys`)
- Systèmes de fichiers en espace utilisateur :
  - Par défaut, les systèmes de fichiers fonctionnent en espace noyau
  - FUSE (*Filesystem in Userspace*) permet d'avoir des systèmes de fichiers en espace utilisateur
    - Pas de besoin de développer à chaque fois un module noyau
    - Ni être root pour insérer le module et faire le montage
  - Exemples :
    - SSHFS : monter un dossier d'une machine distante sur sa machine locale en passant par le protocole SSH
    - Monter une sauvegarde BorgBackup dans un dossier

Types de fichiers

- Fichier ordinaire (suite d'octets)
- Socket
- Lien symbolique
- Périphérique bloc
  - On peut accèder un octet en particulier
  - Exemple : un disque
- Périphérique caractère
  - Manipulation octet par octet, impossible de revenir en arrière
  - Exemple : console série
- Tube
- Dossier

- Donné par la commande / appel-système `stat`
- Affiché dans la sortie de `ls -l`

Fichiers spéciaux

- `/dev/zero` : contient que des zéros
- `/dev/null` : ignore ce qui est écrit dans ce fichier
- `/dev/random` : renvoie des octets aléatoires
- ...

- Ces fichiers peuvent être créés avec la commande `mknod`
- On indique leur comportement en précisant un numéro majeur et mineur
- Ces numéros définis par le noyau permet d'exécuter la fonction correspondant au comportement
  - *cf* https://www.kernel.org/doc/Documentation/admin-guide/devices.txt

Path Lookup

- Trouver le nœud correspondant à un chemin (chaîne de caractères)
- Opération très lente :
  - À chaque élément du chemin :
    - Trouver l'enfant correspondant dans le répertoire courant
    - Vérifier les permissions
  - ⇒ Multiples accès mémoire, accès disques, verrous pris, ...

⇒ Mise en place d'un cache par le noyau, partagé entre tous les processus

Filesystem Hierarchy Standard

- *cf* https://fr.wikipedia.org/wiki/Filesystem_Hierarchy_Standard
- Indique quelle doit être l'organisation de base de certains dossier d'un système GNU/Linux
- Notamment `/bin`, `/dev`, `/etc`, `/home`, `/lib`, `/root`, `/tmp`, `/var`
- Ces dossiers peuvent être des liens symboliques ou des points de montages différents
  - `/bin,sbin,lib/` sont des liens symboliques vers `/usr/bin,sbin,lib/` (*cf* https://systemd.io/THE_CASE_FOR_THE_USR_MERGE/)
  - `/home` est un montage NFS
  - `/var/log` est dans une partition séparée

chroot

- Commande / appel-système pour lancer un processus qui aura pour répertoire racine celui passé en paramètre
- Permet de faire croire aux processus qu'ils sont dans un système normal, alors qu'ils sont dans un « sous- » système
- Exemple : réparer son GRUB cassé depuis un système live :

```
fdisk -l
mount /dev/sda1 /mnt/
mount -t proc none /mnt/proc
mount -o bind /dev /mnt/dev
mount -t sysfs sys /mnt/sys
chroot /mnt/ /bin/bash
update-grub
sync
reboot
```

Systèmes de fichiers distribués

- RAID et ses variantes permettent d'aggréger les disques d'**une** machine
- Mais que se passe-t-il si :
  - la machine entière tombe en panne ?
  - toutes les machines du réseau accèdent aux disques de cette unique machine ?

⇒ Utilisation de systèmes de fichiers *distribués*

- Le même système de fichiers fonctionne sur plusieurs machines
  - Mêmes logiques que le RAID mais au niveau machine : tolérance aux pannes, parallélisation des accès, ...
- L'accès au système de fichiers implique des communications réseau
- Souvent un serveur pour le stockage des méta-données, puis plusieurs serveurs pour le stockage des données
- Exemples : Ceph, GlusterFS, Lustre, MooseFS, BeeGFS, Lustre, ...

Stockage objet

- Utilisée pour stocker des données non-structurées
- Pas d'organisation en hiérarchie
- Chaque objet possède un identifiant unique
- Les objets sont immuables
- Regroupement d'objets aux propriétés similaires (droits) au sein de *buckets*
- Manipulation généralement au travers de protocoles web (HTTP, ...)

- Standard S3 de AWS pour l'interaction
- Serveurs : MinIO, Garage, ...

Virtualisation

Intérêts de la virtualisation

- Exécuter plusieurs OS en même temps sur la même machine
- Serveurs indépendants sur une même machine
  - Bug logiciel dans une machine virtuelle ne gêne pas une autre machine virtuelle
  - Pouvoir redémarrer une machine virtuelle sans impacter les autres
- Développement et débogage
  - Développements d'un noyau sans devoir redémarrer en permanence sa machine physique
  - Pas grave si le système de fichiers est corrompu
- Exécuter sur la même machine physique des logiciels incompatibles entre eux
- Déplacer une machine virtuelle d'une machine physique à une autre
- Réduction des coûts : au lieu d'une machine physique par service, une machine virtuelle par service
  - Économies en matériel, électricité, espace, ...

Terminologie

- **Hôte** (*host*) : machine physique
- **VM** : *Virtual Machine*
- **OS invité** (*guest*) : OS tourant dans une VM
- **Hyperviseur** : logiciel qui fait fonctionner les VMs

Différents types de virtualisation

- Virtualisation totale
- Para-virtualisation

- L'une n'exclut pas l'autre
- Parfois des frontières un peu floues...
  - Les acteurs historiques de la virtualisation totale se mettent à faire de la para-virtualisation
  - Les acteurs historiques de la para-virtualisation se mettent à faire de la virtualisation totale

- Conteneurs

Virtualisation totale

- Exposition de matériel virtuel (processeur, mémoire, disque, réseau, ...)
- L'OS invité est un processus de l'OS hôte
- Chaque instruction de la VM est émulée par l'hôte
  - Réalisé par l'hyperviseur (le processus sur l'OS hôte)
  - En profite pour vérifier les permissions nécessaires (accès mémoire, ...)
  - **Très lent**
- Permet d'exécuter n'importe quel OS invité
  - Du moment que l'OS invité supporte le matériel virtuel exposé
- L'OS invité n'a pas conscience d'être virtualisé

- QEMU, Xen HVM, ...

Virtualisation totale accélérée

- Pas besoin d'émuler *toutes* les instructions de la VM...
- Le processeur hôte peut directement exécuter les instructions non-privilégiées de la VM...
- ... et demander à l'hyperviseur d'émuler les instructions privilégiées

⇒ Beaucoup plus rapide !

- Mais restreint les VMs à avoir la même architecture CPU que l'hôte
- Support matériel :
  - Une exception causée par l'OS invité peut être traitée par l'OS invité et non l'hôte
  - Distinction entre tables de page hôte et invité
  - ...
  - Instructions spécifiques : Intel VT-x, AMD-V
    - À activer dans le BIOS
    - `grep -E '(vmx|svm)' /proc/cpuinfo`
  - On ne fait plus de virtualisation sans

- kQEMU (QEMU avec module Linux KVM), VirtualBox, ...

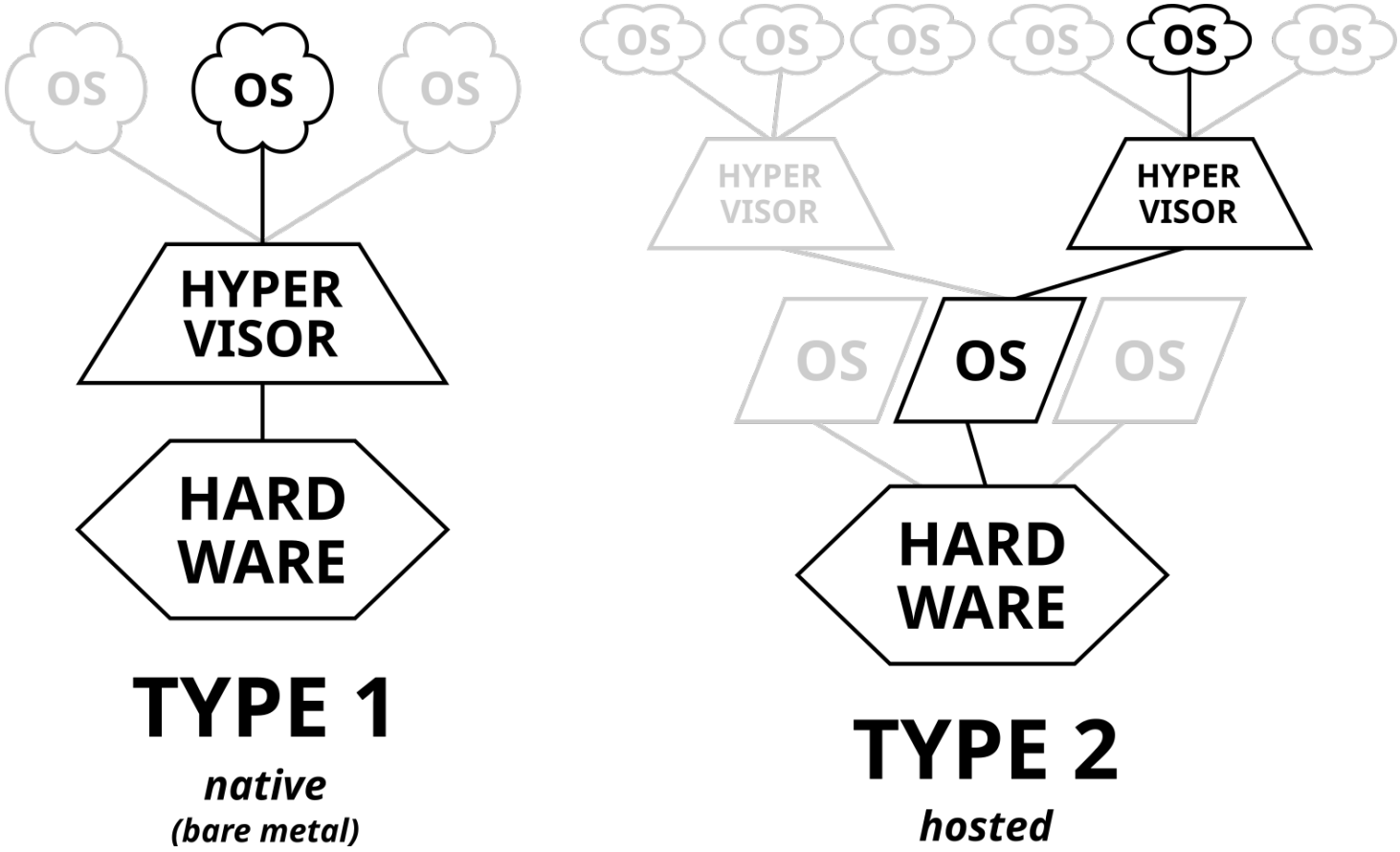
Para-virtualisation

- L'OS invité est conscient qu'il est virtualisé
- Version spécifique du noyau et/ou des drivers dans l'invité
  - Exemple : composant *virtio* sur KVM : driver qui permet de partager des pages mémoires entre invité et hôte
- Manipulation de matériel exposé par l'hyperviseur plus simple

- Historiquement : Xen

Deux types d'hyperviseurs

Même si la catégorisation devient de plus en plus floue...

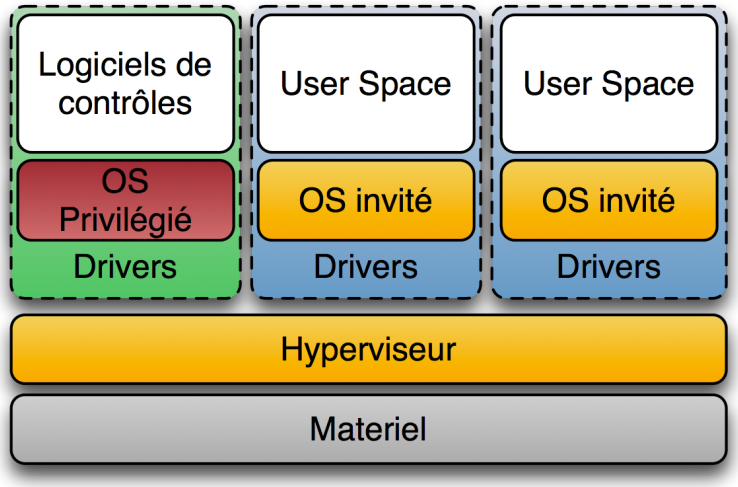


https://fr.wikipedia.org/wiki/Hyperviseur

Deux types d'hyperviseurs

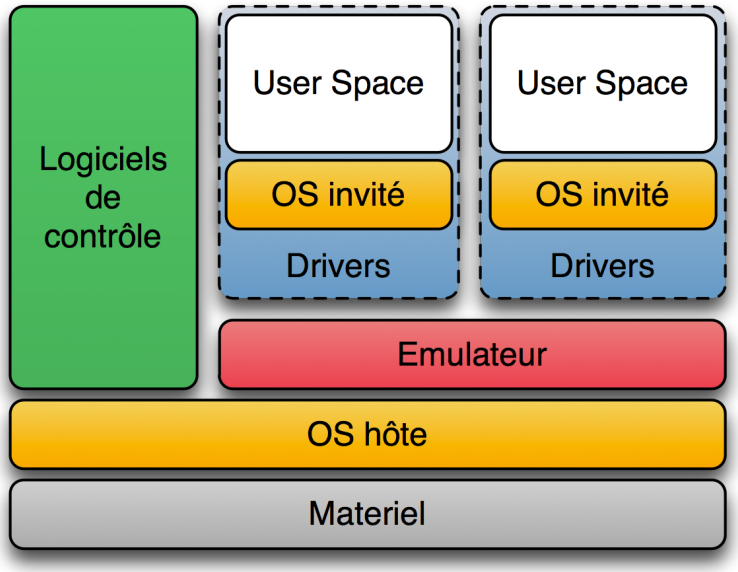
Même si la catégorisation devient de plus en plus floue...

Type 1



- Le processeur exécute un noyau très simplifié (hyperviseur)
- La première VM a plus de droit et permet de gérer les VMs
- Xen

Type 2



- Le processeur exécute un OS « classique », qui fait tourner un émulateur/hyperviseur
- QEMU, VirtualBox, ...

Isolation avec Linux

Processus

- Forme d'isolation rudimentaire (mémoire notamment)
- Mais le processus sait qu'il n'est pas tout seul, qu'il partage des ressources

Cgroups

- Mécanisme pour restreindre les ressources matérielles visibles par un processus (et ses enfants)
- Exemples : seulement un cœur, seulement tant de mémoire RAM, nombre maximum de processus enfants, ...

Namespaces

- Permet de *virtualiser* les ressources logicielles du noyau
- Un type de namespace par ressource : PID, montage, réseau, utilisateur, nom de le machine, ...
- Exemple : on peut mettre un processus dans un namespace qui y aura le PID 1, mais qui aura un autre PID sur le système hôte

Cgroups + namespaces = conteneurs

- Isolation purement logicielle
- Ce n'est plus de la virtualisation
  - Au sens émulation
  - Pas besoin d'émulation, pas besoin de support matériel
- **Les conteneurs partagent / utilisent tous le même noyau de l'hôte !**
  - On ne peut pas tout faire au sein d'un conteneur (charger un module noyau, certaines interactions avec le matériel, ...)
  - Faille/bug dans le noyau ⇒ tous les conteneurs sont impactés
- N'est pas une solution d'isolation **sécurisée** !

- Docker n'a rien inventé
  - Juste une interface plus simple vers les cgroups, namespaces et chroot
  - Notion d'image en plus pour indiquer quels programmes sont présents au sein du conteneur
- Podman, LXC, ...
- *Jail* chez FreeBSD

Logiciels, entreprises, ... de virtualisation

- Beaucoup ont déjà été cités
- VMware : solution logicielle complète de virtualisation, gestion de parc de VM, ...
- Proxmox : l'alternative libre à VMware ?
  - kQEMU et conteneurs
  - Interface web ou CLI
- Xen
  - Historiquement paravirtualisation, type 1
  - Produit commercial : interface de gestion graphique XenServer
  - XCP-ng : distribution Linux (interface web XenOrchestra et CLI)
  - Sinon CLI
- Parallels Desktop : pour hôte MacOS

Conclusion sur la virtualisation

- Objectifs contradictoires :
  - Invité non modifié
  - Vitesse d'exécution
  - Isolation
  - Flexibilité
- Différentes solutions techniques
- Choisir la solution adaptée aux contraintes et objectifs

Ce qui ne rentrait pas dans les autres sections

Nouveautés matérielles

Processeurs hétérogènes

- Un processeur avec des cœurs très peu énergivores (mais peu puissant) et des cœurs puissants (mais énergivores)
- ARM big.LITTLE, mais aussi chez Intel et AMD
- Quels processus exécuter sur quels cœurs ?

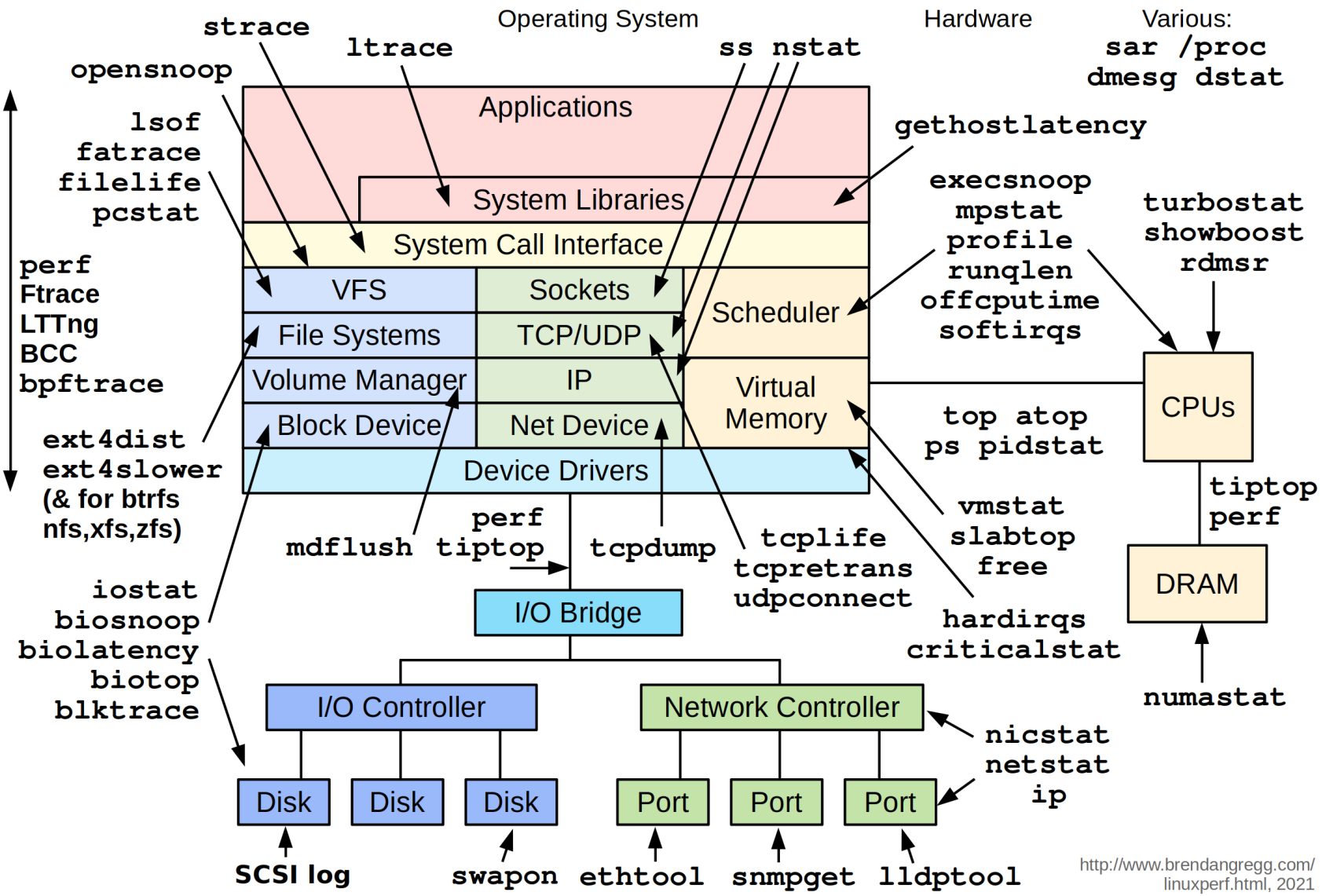
CXL

- *Compute Express Link*
- Standard pour la mémoire *désaggrégée*
- La RAM d'une machine est en réalité sur plusieurs machines
- Facile d'ajouter de la RAM à une VM, même s'il n'y en a plus sur l'hôte
- 200 ns de latence (RAM locale : 50-100 ns)
- Comment le noyau peut l'utiliser ?

Enclaves sécurisées

- Permet matériellement et cryptographiquement d'avoir la garantie que le code exécuté est bien celui qu'on souhaite exécuté, le code exécuté n'a pas été modifié
- Nécessite un support du processeur : Intel SGX (*Software Guard Extensions*), Intel TDX (*Trusted Domain Extensions*), AMD SEV (*Secure Encrypted Virtualization*)

Linux Performance Observability Tools



http://www.brendangregg.com/linuxperf.html, 2021

eBPF

- *extended Berkley Packet Filter*
- Programmé injecté depuis l'espace utilisateur dans le noyau
- Pas besoin de modifier le noyau et/ou recompiler le noyau
- Permet d'instrumenter le noyau
  - Analyser son fonctionnement
  - Débugguer
  - Modifier le comportement
- Exemples d'utilisation : traçage, analyse de flux réseau, filtrer les requêtes DNS au niveau noyau, ...

https://ebpf.io/

De la recherche en systèmes d'exploitation ?

- Vérification formelle
- Heuristiques d'ordonnancement
- Exploiter les nouvelles fonctionnalités matérielles
- Sécurité
- Temps réel
- Répondre aux besoins et contraintes en data-center (virtualisation, accès mémoire et disque, ...)
- ...

Conclusion du cours

- Sujet complexe, mais intéressant
- Chaque section pourrait être un cours dédié
- Lien avec d'autres thématiques : architecture matérielle, ordonnancement, systèmes distribués, sécurité, génie logiciel, ...
- Les questions de systèmes d'exploitation peuvent se retrouver dans d'autres domaines